A complete algorithm to find flows in the one-way measurement model

Niel de Beaudrap *
IQC, University of Waterloo
jdebeaud@iqc.ca

Abstract

This article is the complement to [7], which proves that flows (as introduced by [5]) can be found efficiently for patterns in the one-way measurement model which have non-empty input and output subsystems of the same size. This article presents a complete algorithm for finding flows, and a proof of its' correctness, without assuming any knowledge of graph-theoretic algorithms on the part of the reader. This article is a revised version of [4], where the results of [7] also first appeared.

1 Introduction

In the one-way measurement model [1, 2, 3], algorithms are essentially described by a sequence of single-qubit measurements (where the choice of measurement may depend on earlier measurement results in a straightforward way) performed on a many-qubit entangled state. This many-qubit state may be described in terms of the state of an *input system* I, together with a graph G of *entangling operations* involving I and a collection of auxiliary qubits prepared in the $|+\rangle$ state: each edge of G represents a single controlled-Z operation between two qubits. After the sequence of measurements, any qubits left unmeasured still support a quantum state, and are interpreted as an *output system* O. A triple (G, I, O) belonging to a given pattern is called the *geometry* of the pattern.

In [4], it was shown that the flow property defined by Danos and Kashefi [5] can be efficiently tested for a geometry (G, I, O) when $|I| = |O|$. The property is the existence of a *causal flow*[1], which describes a partial order $\preccurlyeq$ describing an order (independent of measurement angles) in which the qubits of the geometry may be measured to perform a unitary embedding, once suitable corrections are applied to the output qubits. Causal flows may allow quantum algorithms to be devised in the one-way measurement model without using the circuit model: [6] proposes one way in which this might be done.

This article presents a complete algorithm for finding causal flows for a geometry (G, I, O) with $|I| = |O|$ in time $O(km)$, where $k = |I| = |O|$ and $m = |E(G)|$, suitable for an audience with no experience in graph-theoretic algorithms. This is a revised version of [4], re-written with the aim of focusing on the algorithm for finding flows for the sake of reference. For the graph-theoretical characterization of flows, this article refers to [7], which is an improved presentation of the graph-theoretic results presented originally in [4].

Although no knowledge of graph-theoretic algorithms is assumed, a basic understanding of graph theory and the one-way measurement model is essential. For basic definitions in graph theory, readers may refer to Diestel's excellent text [9]; I will use the conventions of [7, 8] for describing patterns in the one-way model.

2 Preliminaries

In this section, we will fix our conventions and review the results and terminology of [7].

*This work was supported in part by ARDA, ORDCF, MITACS, and CIAR.
[1] These are simply called "flows" in [5]: I use the term "causal flow" in this article to maintain consistency with [7].

2.1 Basic notation and conventions

For a graph G, we write $V(G)$ for the set of vertices and $E(G)$ for the set of edges of G. Similarly, for a directed graph (or *digraph*) D, we write $V(D)$ for the set of vertices and $A(D)$ for the set of directed edges (or *arcs*) of D. If x and y are adjacent, we let xy denote the edge between them in a graph, and $x \to y$ denote an arc from x to y in a digraph. We will use the convention that digraphs may contain loops on a single vertex and multiple edges between two vertices, but that graphs cannot have either.

When a graph G is clear from context, we will write $x \sim y$ when x and y are adjacent in G, and write S^{c} to represent the complement of a set of vertices $S \subseteq V(G)$.

If $\mathcal{C}$ is a collection of directed paths (or *dipaths*), we will say that $x \to y$ is an *arc of* $\mathcal{C}$, and that the edge xy is *covered* by $\mathcal{C}$, when $x \to y$ is an arc in a path $P \in \mathcal{C}$.

In this paper, $\mathbb{N}$ denotes the non-negative integers. For any $n \in \mathbb{N}$, $[n]$ denotes the set $\{j \in \mathbb{N} \mid j < n\}$.

2.2 Results for Causal Flows

2.2.1 Definition and motivation

Definition 1. A *geometry* (G, I, O) is a graph G together with subsets $I, O \subseteq V(G)$. We call I the *input vertices* and O the *output vertices* of the geometry. A *causal flow* on (G, I, O) is an ordered pair $(f, \preccurlyeq)$, with a function $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$ and a partial order $\preccurlyeq$ on $V(G)$, such that

$$(\mathsf{F}i) \quad x \sim f(x); \qquad (\mathsf{F}ii) \quad x \preccurlyeq f(x); \qquad (\mathsf{F}iii) \quad y \sim f(x) \implies x \preccurlyeq y, \tag{1}$$

hold for all vertices $x \in O^{\mathsf{c}}$ and $y \in V(G)$. We will refer to f as the *successor function* of the causal flow, and $\preccurlyeq$ as the *causal order* of the causal flow.

A geometry (G, I, O) represents the information of a one-way measurement pattern which is independent of the order of operations and measurement angles. G is the entanglement graph of the pattern, I is the set of qubits which are not prepared in a fixed state initially (their joint initial state in the algorithm may be arbitrary), and O represents the set of qubits which are not measured in the pattern (which thus support a final quantum state).

The conditions $(\mathsf{F}i)$ – $(\mathsf{F}iii)$ are motivated by how byproduct operators and signal dependencies are induced by commuting correction operations to the end of a pattern which performs a unitary embedding. The significance of a causal flow on a geometry (G, I, O) is that any pattern defined on that geometry can be transformed into one which has the same measurement angles and which performs a unitary embedding $\mathcal{H}_I \longrightarrow \mathcal{H}_O$. In particular, this means that unitary embeddings can be devised in the measurement model by ignoring signal dependencies and treating each measurement operator as though it post-selects for some one of the states in the basis of the measurement. See Section 2.2 of [7] for details.

2.2.2 Graph-theoretic characterization

The result of [7] was obtained by characterizing causal flows in terms of collections of vertex-disjoint paths.

Definition 2. Let (G, I, O) be a geometry. A collection $\mathcal{C}$ of (possibly trivial) directed paths in G is a *path cover* of (G, I, O) if

- (i). each $v \in V(G)$ is contained in exactly one path (i.e. the paths cover G and are vertex-disjoint);
- (ii). each path in $\mathcal{C}$ is either disjoint from I, or intersects I only at its initial point;
- (iii). each path in $\mathcal{C}$ intersects O only at its final point.

The *successor function* of a path cover $\mathcal{C}$ is the unique $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$ such that $y = f(x)$ if and only if $x \to y$ is an arc of $\mathcal{C}$. If a function $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$ is a successor function of *some* path-cover of (G, I, O), we call f a *successor function of* (G, I, O).

If a geometry (G, I, O) has a causal flow $(f, \preccurlyeq)$, the maximal orbits of the successor function f define a path cover for (G, I, O), which allow us to consider the causal flow in terms of vertex-disjoint paths in G:

Theorem 3 [7, Lemma 3]. *Let $(f, \preccurlyeq)$ be a causal flow on a geometry (G, I, O). Then there is a path cover $\mathcal{P}_f$ of (G, I, O) whose successor function is f.*

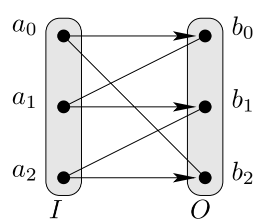
Given that the successor function of a causal flow for (G, I, O) induces a path cover, one might think of also trying to obtain a causal flow from the successor function of a path cover. There is an obvious choice of binary relation for a successor function f:

Definition 4. Let f be a successor function for (G, I, O). The *natural pre-order*[2] $\preccurlyeq$ *for* f is the transitive closure on $V(G)$ of the conditions

$$x \preccurlyeq x \; ; \qquad\qquad x \preccurlyeq f(x) \; ; \qquad\qquad y \sim f(x) \implies x \preccurlyeq y \;, \tag{2}$$

for all $x, y \in V(G)$.

If $\preccurlyeq$ is a partial order, it will be the coarsest partial order such that $(f, \preccurlyeq)$ is a causal flow. However, it is easy to construct geometries where $\preccurlyeq$ is not a partial order. Figure 1 illustrates one example. For any choice of successor function f on this geometry, (F*iii*) forces either $a_0 \preccurlyeq a_1 \preccurlyeq a_2 \preccurlyeq a_0$ or $a_0 \succcurlyeq a_1 \succcurlyeq a_2 \succcurlyeq a_0$ to hold. Because a_0, a_1, and a_2 are distinct, such a relation $\preccurlyeq$ is not antisymmetric, so it isn't a partial order.



$$(\mathsf{F}ii) \implies \left\{ \begin{array}{c} a_0 \preccurlyeq b_0 \\ a_1 \preccurlyeq b_1 \\ a_2 \preccurlyeq b_2 \end{array} \right\}$$

$$(\mathsf{F}iii) \implies a_0 \preccurlyeq a_1 \preccurlyeq a_2 \preccurlyeq a_0$$

Figure 1: A geometry with a successor function $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$, but no causal flow.

In the example above, we have a cycle of relationships induced by condition (F*iii*). The following definitions characterize when such cycles of relationships occur.

Definition 5. Let (G, I, O) be a geometry, and $\mathcal{F}$ a family of directed paths in G. A walk $W = u_0 u_1 \cdots u_\ell$ is an *influencing walk*[3] for $\mathcal{F}$ if it is a concatenation of zero or more paths (called *segments* of the influencing walk) of the following two types:

- xy, where $x \to y$ is an arc of $\mathcal{F}$;
- xzy, where $x \to z$ is an arc of $\mathcal{F}$ and $yz \in E(G)$.

A *vicious circuit* for $\mathcal{F}$ is a closed influencing walk for $\mathcal{F}$ with at least one segment.

Theorem 6 [7, Lemma 9]. *Let $\mathcal{C}$ be a path cover for (G, I, O) with successor function f, and let $\preccurlyeq$ be the natural pre-order of f. Then $x \preccurlyeq y$ if and only if there is an influencing walk for $\mathcal{C}$ from x to y.*

Given that we want to forbid cycles of relationships for the natural pre-order $\preccurlyeq$, we are then interested in the following restriction of path covers:

[2] A pre-order is a binary relation which is reflexive and transitive, but not necessarily antisymmetric.
[3] These are closely related to *walks which alternate with respect to $\mathcal{F}$*: see Section 3.1.1.

Definition 7. A path cover $\mathcal{C}$ for (G, I, O) is a *causal path cover* if $\mathcal{C}$ does not have any vicious circuits in G.

Theorem 8 [7, Theorem 10]. *Let (G, I, O) be a geometry with path cover $\mathcal{C}$, f be the successor function of $\mathcal{C}$, and $\preccurlyeq$ be the natural pre-order for f. Then $\mathcal{C}$ is a causal path cover if and only if $\preccurlyeq$ is a partial order, which occurs if and only if $(f, \preccurlyeq)$ is a causal flow for (G, I, O).*

By characterizing causal flows in terms of causal path covers, we can make use of the following result:

Theorem 9 [7, Theorem 11]. *Let (G, I, O) be a geometry such that $|I| = |O|$, and let $\mathcal{C}$ be a path cover for (G, I, O). If $\mathcal{C}$ is a causal path cover, then $\mathcal{C}$ is the only maximum collection of vertex-disjoint $I - O$ dipaths.*

Then, if $|I| = |O|$ and (G, I, O) has a causal flow, there is a unique maximum-size collection of vertex-disjoint $I - O$ paths, and that collection is a causal path cover which allows one to reconstruct a causal flow. Taking the contrapositive, if we can find a maximum-size collection of vertex-disjoint paths from I to O which is not a causal path cover, then (G, I, O) does not have a causal flow.

3 An efficient algorithm for finding a causal flow when $|I| = |O|$

Using Theorems 8 and 9 when $|I| = |O|$, we can reduce the problem of finding a causal flow to finding a maximum-size family of vertex-disjoint I–O paths in G. Given such a family of paths $\mathcal{F}$, we may then verify that the resulting family forms a path cover for G, obtain the successor function f of $\mathcal{F}$, and attempt to build a causal order compatible with f. We illustrate how this may efficiently be done in this section.

Implementation details. For the purpose of run-time analysis, I fix here conventions for the data structures used to implement graphs, paths, and sets throughout the following algorithms.

- We will assume an implementation of graphs and digraphs using adjacency lists for each vertex x (in the case of digraphs, using two separate lists for the arcs entering x and those leaving x). Such an implementation can be easily performed in space $O(m)$, where m is the number of arcs/edges, assuming a connected (di-)graph.[4]
- Sets of vertices are considered to be implemented via arrays storing the characteristic function of the set. We may assume without loss of generality that these are also used to perform bounds-checking on arrays which are used to implement partial functions on $V(G)$, such as successor functions $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$.
- Collections of vertex-disjoint di-paths $\mathcal{F}$ in a graph G will be implemented as a set $V(\mathcal{F})$ indicating for each $x \in V(G)$ whether x is covered by $\mathcal{F}$, and a digraph containing all of the arcs of $\mathcal{F}$. As well, functions `prev` and `next` will be defined for all vertices in I^{c} (respectively, O^{c}) covered by $\mathcal{F}$ which returns the predecessor (respectively, successor) of a vertex covered by $\mathcal{F}$.

  Throughout some of the algorithms below, a family of vertex-disjoint paths may be transformed into to a graph where a single vertex has out-degree 2, but every other vertex has out-degree at most 1, and every vertex has in-degree at most 1. So long as these bounds are maintained, determining whether a vertex is covered by $\mathcal{F}$, whether an arc is in $\mathcal{F}$, and adding/deleting arcs from $\mathcal{F}$ can be done in constant time. As well, the function `prev` will be well-defined so long as the in-degree of the graph representation of $\mathcal{F}$ is bounded by 1.

[4] This holds, in particular, for graphs corresponding to one-way patterns implementing unitary operations which are not tensor-product decomposable.

3.1 Efficiently finding a path cover for (G, I, O)

Given a geometry (G, I, O), we are interested in obtaining a maximum-size family $\mathcal{F}$ of disjoint I–O paths in G in order to test whether it is a causal path cover. This is known to be efficiently solvable.

Problems involving constructing collections of paths with some extremal property in graphs are usually solved by reducing the problem to a problems of network flows on digraphs: algorithms for such problems have been very well studied. (Section 4.1 of [7] outlines an algorithm of this kind to find a maximum-size family of disjoint $I - O$ paths.) However, in order to present a solution which does not assume any background in graph-theoretic algorithms, and also in order to reduce the number of auxiliary concepts involved in the solution, I will present an algorithm not explicitly based on network flows.[5] A dividend of such a presentation is that it highlights the relationship between influencing walks and *walks which alternate with respect to a collection of disjoint paths*, which was alluded to in Definition 5.

3.1.1 Alternating and augmenting walks

Definition 10. Let $I, O \subseteq V(G)$. A collection of vertex-disjoint paths from I to O is *proper* if its' paths intersect I and O only at their endpoints.

A collection of k vertex-disjoint $I - O$ paths of is necessarily proper when $|I| = |O| = k$. We would like to arrive at such a maximum-size collection by producing successively larger proper collections of vertex-disjoint paths. To so so, we will use results of graph theory pertaining to Menger's Theorem. The basic approach present is outlined in Section 3.3 of [9].

Definition 11. For a family $\mathcal{F}$ of vertex-disjoint directed paths from I to O, a walk $W = u_0 u_1 \cdots u_\ell$ in G is said to be *pre-alternating with respect to* $\mathcal{F}$ if the following hold for all $0 < j, k \leqslant \ell$:

(i). $\mathcal{F}$ does not contain $u_j \to u_{j+1}$ as an arc;

(ii). if $u_j = u_k$ and $j \neq k$, then u_j is covered by $\mathcal{F}$;

(iii). if u_j is covered by $\mathcal{F}$, then either $u_j \to u_{j-1}$ or $u_{j+1} \to u_j$ is an arc of $\mathcal{F}$.

W is said to be *alternating with respect to* $\mathcal{F}$ if W is pre-alternating with respect to $\mathcal{F}$, and u_0 is an element of I not covered by $\mathcal{F}$. W is an *augmenting walk* for $\mathcal{F}$ if W alternates with respect to $\mathcal{F}$, and $u_\ell \in O$.

Figure 2 illustrates two pre-alternating walks for a family $\mathcal{F}$ of vertex-disjoint paths in a geometry (G, I, O).

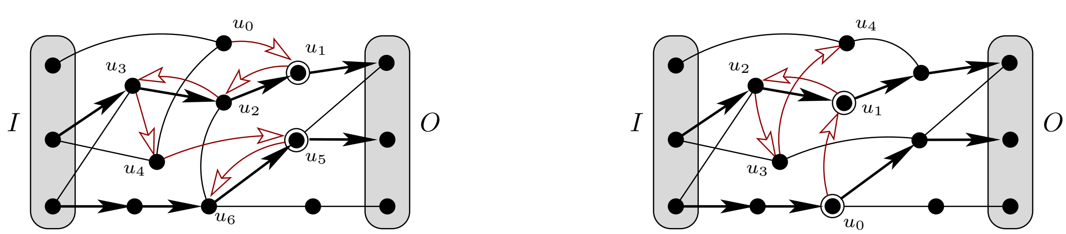


Figure 2: Two examples of a walk W (hollow arrows) which is pre-alternating with respect to a collection $\mathcal{F}$ of vertex-disjoint paths from I to O (solid arrows). In both examples, circled vertices are *entry points* of W into $\mathcal{F}$ (see Definition 13).

The relationship between influencing walks and pre-alternating walks is most clear for a path cover $\mathcal{C}$ of (G, I, O), in which case an influencing walk for $\mathcal{C}$ is the reverse of a walk which is pre-alternating for $\mathcal{C}$. As we will see in the next few pages, pre-alternating walks describe ways in which different families of disjoint paths from I to O are related to each other: this is essentially the reason why a vicious circuit (i.e. a closed influencing walk) exists for a path cover whenever there is a second family of disjoint $I - O$ paths of the same size.

[5] The solution presented here can be easily related to the solution via network flows, but a small amount of additional work must be done in order to stay in the context of collections of disjoint paths, rather than disjoint paths, cycles, and walks of length 2.

First, we will show that augmenting walks for $\mathcal{F}$ are always present if $|\mathcal{F}| < |I| = |O|$, and if there is a family of vertex-disjoint $I - O$ paths of size k:

Theorem 12. *Let G be a graph, and $I, O \subseteq V(G)$ with $|I| = |O| = k$. Let $\mathcal{F}$ be a collection of vertex-disjoint $I - O$ paths with $|\mathcal{F}| < k$, and $i \in I$ be a vertex not covered by $\mathcal{F}$. If there is a collection $\mathcal{C}$ of vertex-disjoint dipaths from I to O with $|\mathcal{C}| = k$, then there is an augmenting walk W for $\mathcal{F}$ starting at i which traverses each edge of G at most once, and where i is the only input vertex in W not covered by $\mathcal{F}$.*

Proof — Suppose G contains a collection $\mathcal{C}$ of k vertex-disjoint $I - O$ dipaths, let $\mathcal{F}$ be some proper collection of vertex-disjoint $I - O$ dipaths of size less than k, and let $I' \neq \varnothing$ be the set of input vertices not covered by $\mathcal{F}$. Let us say that a vertex $v \in V(G)$ is an *incidence point* of $\mathcal{C}$ and $\mathcal{F}$ if v is covered by both $\mathcal{C}$ and $\mathcal{F}$, and there is a vertex w which is adjacent to v in a path of $\mathcal{C}$ but which is not adjacent to v in any path of $\mathcal{F}$. Let $\mathcal{I}$ be the set of incidence points of $\mathcal{C}$ and $\mathcal{F}$, and let $\mathcal{G}$ be a di-graph with $V(\mathcal{G}) = I' \cup \mathcal{I} \cup O$, and $(x \to y) \in A(\mathcal{G})$ for $x, y \in V(Q)$ if one of the following applies:

- there exists a vertex $z \in \mathcal{I}$ such that
  (i). x and z lie on a common path P in $\mathcal{C}$, where z is the next incidence point in P after x, and
  (ii). y and z lie on a common path P' in $\mathcal{F}$, where z is the next incidence point in P' after y;
- x and y lie on a common path P in $\mathcal{C}$, there are no incidence points on P after x, and $y \in O$.

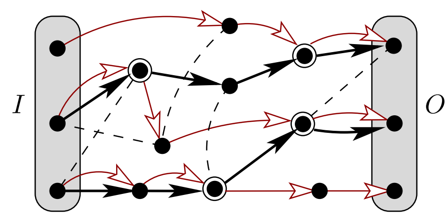


Figure 3: Two families of vertex-disjoint $I - O$ paths in a graph: one family $\mathcal{C}$ with k paths (hollow arrows), and one family $\mathcal{F}$ with $< k$ paths (solid arrows). Circled vertices are the incidence points of $\mathcal{C}$ and $\mathcal{F}$. Dashed lines are the other edges of the graph.

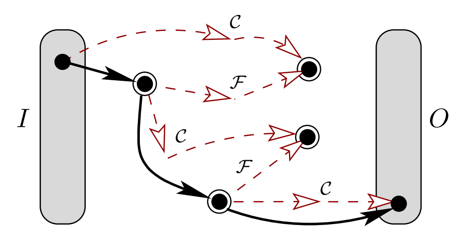


Figure 4: The digraph $\mathcal{G}$ obtained by applying the construction above to Figure 3. Dashed arrows represent the edges from path segments belonging to either $\mathcal{C}$ or $\mathcal{F}$ in the original graph; thick black arrows are the actual arcs of $\mathcal{G}$, which are induced by those path segments.

Because both $\mathcal{C}$ and $\mathcal{F}$ are vertex-disjoint collections of paths, it is easy to show that the maximum in-degree and out-degree of $\mathcal{G}$ are both 1. Thus, $\mathcal{G}$ consists of vertex-disjoint di-cycles, walks of length 2, isolated vertices, and directed paths.

Because each $v \in I'$ is not covered by a path of $\mathcal{F}$, and is not preceded by any vertices in it's respective path of $\mathcal{C}$, it has in-degree 0 in $\mathcal{G}$. Then, each element of I' is at the beginning of a maximal dipath in $\mathcal{G}$. Furthermore, each vertex in $v \in I' \cup \mathcal{I}$ has out-degree 1: if $P \in \mathcal{C}$ is the path covering v, either there are no incidence vertices after v on P, in which case there is an arc $v \to y$ for the vertex $y \in O$ at the end of P; or if we let $z \in \mathcal{I}$ be the first incidence vertex following v on P, we will have $z \notin I$, in which case there will be an incidence vertex w which precedes z on some path of $\mathcal{F}$, because all input vertices covered by $\mathcal{F}$ are incidence points. Thus, any maximal di-path in $\mathcal{G}$ must end in O. Then, for each $i \in I'$, there is a dipath from i to some element of O in the graph $\mathcal{G}$.

Consider any vertex $i \in I'$, and let $u_0 \to u_1 \to \cdots \to u_\ell$ be the dipath in $\mathcal{G}$ from i to O. Let $P \in \mathcal{C}$ and $P' \in \mathcal{F}$ be the paths containing $u_{\ell-1}$: from $(u_{\ell-2} \to u_{\ell-1}) \in A(\mathcal{G})$, we know that there is an incidence vertex after $u_{\ell-1}$ in the path P'. Note that u_ℓ is either not covered by any path of $\mathcal{F}$, or it occurs at the end of a path of $\mathcal{F}$ and is not followed by any vertices on that path; then, $(u_{\ell-1} \to u_\ell) \in A(\mathcal{G})$ implies that there are no incidence points on P after $u_{\ell-1}$. Then, the arcs leaving $u_{\ell-1}$ in P and P' must be different: the fact that no incidence point follows $u_{\ell-1}$ in P then implies that no path of $\mathcal{F}$ intersects P after $u_{\ell-1}$. In particular, u_ℓ is not covered by $\mathcal{F}$.

Because $u_0 \in I'$ and $u_\ell \in O$ are both not covered by $\mathcal{F}$, we may construct an augmenting walk W for $\mathcal{F}$ in the original graph G, as follows. If $\ell = 0$, we let W be the trivial path on u_0, which is an augmenting walk for $\mathcal{F}$. Otherwise:

- For each $j \in [\ell - 1]$, let v_j be the next incidence point after u_j on the path $P_j \in \mathcal{C}$ containing u_j. (This v_j will then also be the next incidence point after u_{j+1} on the path $P'_j \in \mathcal{F}$ containing u_{j+1}.)

- Let $\tilde{P}_j$ be the segment of P_j from u_j to v_j, and $\tilde{P}'_j$ be the *reverse* of the segment of P'_j from u_{j+1} to v_j.

- Finally, let $\tilde{P}_{\ell-1}$ be the path segment in $\mathcal{C}$ from $u_{\ell-1}$ to u_ℓ.

Then, define $W = u_0 \tilde{P}_0 v_0 \tilde{P}'_0 u_1 \tilde{P}_1 \cdots u_{\ell-1} \tilde{P}_{\ell-1} u_\ell$. We may show that W is an augmenting walk for $\mathcal{F}$:

(i). Each path $\tilde{P}_j$ is internally disjoint from $\mathcal{F}$, because they are sub-paths of elements of $\mathcal{C}$, and do not contain any incidence points in their interiors. Then, none of the arcs of $\tilde{P}_j$ are arcs of $\mathcal{F}$ for any $j \in [\ell]$. Also, all of the arcs of the paths $\tilde{P}'_j$ are the reverse of arcs of $\mathcal{F}$: they do not contain arcs of $\mathcal{F}$ either. Then, none of the arcs of W are arcs of $\mathcal{F}$.

(ii). Because $u_0 \to \cdots \to u_\ell$ is a directed path in $\mathcal{G}$, we have $u_j \neq u_k$. Because each path $\tilde{P}_j$ and $\tilde{P}'_j$ is uniquely determined by u_j for $j \in [\ell - 1]$, those sequences of vertices can also occur only once each. Each interior vertex of $\tilde{P}_j$ or $\tilde{P}'_j$ can only occur in a single path of $\mathcal{C}$ or $\mathcal{F}$, between two consecutive elements of $I' \cup \mathcal{I} \cup O$ on that path: then, because each segment $\tilde{P}_j$ and $\tilde{P}'_j$ only occur once in W, each interior vertex of those segments also occurs only once in W.

Thus, if any vertex x occurs more than once in W, x must be an end point of some path $\tilde{P}_j$ or $\tilde{P}'_j$. Aside from $\tilde{P}_0$ and $\tilde{P}_{\ell-1}$, both end-points of each such segment has in-degree 1 and out-degree 1, so they cannot be elements of either I' or O. Then, any vertex which occurs more than once in W is an element of $\mathcal{I}$, and is therefore covered by $\mathcal{F}$.

(iii). The only points in W which are covered by $\mathcal{F}$ are the vertices of the paths $\tilde{P}_j$ for $j \in [\ell - 1]$, which are all at the beginning or the end of arcs in W which are the reverse of arcs of $\mathcal{F}$.

Thus, W is an augmenting walk for $\mathcal{F}$. Furthermore, because each edge of G is contained in at most one segment $\tilde{P}_j$ or $\tilde{P}'_j$, each edge occurs at most once in W. Finally, because elements of I' have in-degree 0 in $\mathcal{G}$ and do not occur in the segments $\tilde{P}_j$ or $\tilde{P}'_j$, any input vertices other than $i = u_0$ which occur on W must be incidence points, which means they are covered by $\mathcal{F}$. Thus, there is a proper augmenting path for $\mathcal{F}$ of the desired type starting at $i \in I$. □

The above Theorem illustrates how we can build an augmenting walk for $\mathcal{F}$ from a collection of disjoint $I - O$ paths which covers I and O. If we impose restrictions on the type of augmenting walk we consider, we may also efficiently do the reverse. The restriction we are interested in is the following:

Definition 13. Let $W = u_0 u_1 \cdots u_\ell$ be a walk which which is pre-alternating with respect to $\mathcal{F}$.

- An *entry point* of W into $\mathcal{F}$ is a vertex u_j which is covered by $\mathcal{F}$, where either $j = 0$ or $u_j \to u_{j-1}$ is not an arc of $\mathcal{F}$.

- The walk W is *monotonic* if, for every path $P \in \mathcal{F}$ and for any indices $0 \leqslant h < j < \ell$ such that u_h and u_j are both entry points for W into $\mathcal{F}$ which lie on P, u_h is closer to the initial point of P than than u_j is.

- W is a *proper* pre-alternating walk if W traverses each edge at most once, each input vertex in W (except possibly u_0) is covered by $\mathcal{F}$, and W is monotonic.

We will be most interested in proper augmenting walks, which are useful in increasing the size of proper collections of $I - O$ paths. The sort of augmenting walk that is guaranteed by Theorem 12 is almost a proper augmenting walk, and merely lacks a guarantee of monotonicity. However, the following Lemma shows that we lose no generality in imposing monotonicity as a condition:

Lemma 14. *Let G be a graph, and $I, O \subseteq V(G)$. Let $\mathcal{F}$ be a collection of vertex-disjoint I – O paths, and let W be an augmenting walk for $\mathcal{F}$ from $i \in I$ to $\omega \in O$. Then there is a monotonic augmenting walk $\mathcal{W}$ for $\mathcal{F}$ from i to ω.*

Proof — Let W be given by $W = u_0 \cdots u_\ell$, where $u_0 = i$ and $u_\ell = \omega$. For any path $P \in \mathcal{F}$, and two entry points u_h and u_j of W into $\mathcal{F}$, let us say that (u_h, u_j) is a *reversed pair* if $h < j$ but u_j is closer to the initial point of P than u_h. We will produce a monotonic augmenting walk by recursively reducing the number of reversed pairs of W.

- If W has no reversed pairs, then W is already monotonic, in which case we may let $\mathcal{W} = W$.
- Suppose that (u_h, u_j) is a reversed pair of W. Then $h < j$, but u_j is closer than u_h to the initial point of the path $Q \in \mathcal{F}$ which covers both of them. Note that $u_\ell = \omega$ is not covered by $\mathcal{F}$, and so is not on the path Q: then, let $j' \in [\ell]$ be the smallest index such that $u_{j'+1}$ is not on Q. Let $Q = q_0 \cdots q_a q_{a+1} \cdots q_{b-1} q_b \cdots q_m$, where $q_a = u_{j'}$ and $q_b = u_h$. Then, let
$$W' = u_0 \cdots u_{h-1} q_b q_{b-1} q_{b-2} \cdots q_{a+1} q_a u_{j'+1} \cdots u_\ell .$$
From the fact that W is an augmenting walk for $\mathcal{F}$, it is easy to show that W' is also an augmenting walk for $\mathcal{F}$. As well, the entry points of W' into $\mathcal{F}$ are a subset of the entry points of W into $\mathcal{F}$, in which case the reversed pairs of W' are also a subset of the reversed pairs of W; and W' does not have (u_h, u_j) as a reversed pair. Then, W' has strictly fewer reversed pairs than W.

Because W is a finite walk, it can have only finitely many reversed pairs; then, by recursion, we may construct a monotonic augmenting walk $\mathcal{W}$ for $\mathcal{F}$ from i to ω. □

Corollary 15. *Suppose $|I| = |O| = k$, $\mathcal{F}$ a proper collection of vertex-disjoint I – O paths in G with $|\mathcal{F}| < k$, and let $i \in I$ be a vertex not covered by $\mathcal{F}$. If there is a collection $\mathcal{C}$ of vertex-disjoint dipaths from I to O with $|\mathcal{C}| = k$, then there is a proper augmenting walk W for $\mathcal{F}$ starting at i.*

Proof — Theorem 12 and Lemma 14. □

For proper augmenting walks, the reason for requiring that no edge is traversed twice is essentially to help construct efficient algorithms for finding them, which we consider later. The requirements that the only input vertex in the walk which is not covered by $\mathcal{F}$, and that it be monotonic, are essentially chosen to allow us to use augmenting walks to increase the size of a *proper* collection of vertex-disjoint paths to cover exactly one more input vertex. We may do this using the following operation:

Definition 16. Let $\mathcal{F}$ be a proper collection of vertex-disjoint I – O dipaths in G, and W be a proper augmenting walk for $\mathcal{F}$. Then, $\mathcal{F} \oplus W$ denotes the collection of directed paths which are formed by those arcs $x \to y$ which belong either to W or a path of $\mathcal{F}$, and for which $y \to x$ is not an arc of either W or $\mathcal{F}$.

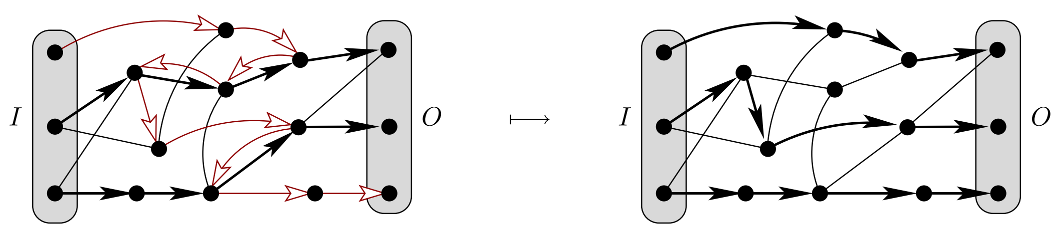


Figure 5: On the left: an example of a proper collection $\mathcal{F}$ of vertex disjoint I – O paths (solid arrows) with a proper augmenting walk W for $\mathcal{F}$ (hollow arrows). On the right: the augmented collection of paths $\mathcal{F} \oplus W$.

The collection $\mathcal{F} \oplus W$ described above is formed by the usual procedure for augmenting a network-flow with an augmenting path: one can think of forming $\mathcal{F} \oplus W$ by "adding" together the arcs of $\mathcal{F}$ and W, and "cancelling" them whenever they point in opposite directions on a single edge.

Lemma 17. *Let $\mathcal{F}$ be a proper collection of vertex-disjoint I – O dipaths in G, and W be a proper augmenting walk for $\mathcal{F}$. Then $\mathcal{F} \oplus W$ is a proper collection of vertex-disjoint I – O dipaths, with $|\mathcal{F} \oplus W| = |\mathcal{F}| + 1$; and the input vertices covered by $\mathcal{F} \oplus W$ are those covered by $\mathcal{F}$ and W together.*

Proof — We induct on the number of times r that the walk W intersects the paths of $\mathcal{F}$. If $r = 0$, then $\mathcal{F} \oplus W = \mathcal{F} \cup \{W\}$, and the inputs covered by $\mathcal{F} \oplus W$ are clearly those covered by $\mathcal{F}$ or by W. Otherwise, suppose that the proposition holds for all cases where the augmenting walk intersects the paths of its' respective collection fewer than r times.

Let W be given by $W = u_0 u_1 \cdots u_a u_{a+1} \cdots u_{b-1} u_b \cdots u_\ell$, where none of the points u_j are covered by $\mathcal{F}$ for $j \in [a]$, and where $u_{j+1} \to u_j$ is an arc of $\mathcal{F}$ for all $a \leqslant j < b$. Let $Q \in \mathcal{F}$ be the path containing u_a through u_b: in particular, let $Q = q_0 q_1 \cdots q_c q_{c+1} \cdots q_{d-1} q_d \cdots q_m$, where $q_c = u_b$ and $q_d = u_a$. Then, we may define

$$Q' = u_0 u_1 \cdots u_a q_{d+1} \cdots q_m \quad , \qquad\qquad W' = q_0 q_1 \cdots q_c u_{b+1} \cdots u_\ell \quad :$$

then $Q' \in \mathcal{F} \oplus W$, and W' is an augmenting walk for $\mathcal{F}' = (\mathcal{F} \setminus Q) \cup \{Q'\}$ which intersects the paths of $\mathcal{F}'$ fewer than r times. Because $\mathcal{F}$ is a proper collection of vertex-disjoint I – O paths, Q' only intersects I and O at its' endpoints, and Q' does not intersect any paths of $\mathcal{F} \setminus Q$, $\mathcal{F}'$ is proper. Similarly, because Q only intersects I at q_0 and because W only intersects I at u_0 and at input vertices covered by $\mathcal{F}$, W' does not cover any inputs except those covered by $\mathcal{F}$. Because W doesn't traverse any edges twice, and all of the other entry points q_h of W into $\mathcal{F}$ on the path Q have $h > c$ by the monotonicity of W, W' itself does not traverse any edge twice. Finally, all of the entry points of W into $\mathcal{F}$ are also entry points of W' into $\mathcal{F}'$, except for u_a: all the other are left unaffected, including the order in which they occur. Then W' is monotonic, so that W' is a proper augmenting walk for $\mathcal{F}'$.

By the induction hypothesis, $\mathcal{F}' \oplus W'$ is a proper collection of vertex-disjoint paths from I to O, with $|\mathcal{F}' \oplus W'| = |\mathcal{F}'| + 1 = |\mathcal{F}| + 1$. Also by induction, the input vertices covered by $\mathcal{F}' \oplus W'$ are those covered by $\mathcal{F}'$ or by W'. Because W' covers the input q_0, and $\mathcal{F}'$ covers all inputs covered by W or by $\mathcal{F}$ except for q_0, $\mathcal{F}' \oplus W'$ then covers all vertices covered by $\mathcal{F}$ or by W. Finally, note that the set of arcs from $\mathcal{F}'$ and W' together only differs from the set of arcs from $\mathcal{F}$ and W together by the absence of the arcs $u_j \to u_{j+1}$ from W and the arcs $u_{j+1} \to u_j$, for $a \leqslant j < b$, which oppose each other. We then have $\mathcal{F}' \oplus W' = \mathcal{F} \oplus W$: thus, $|\mathcal{F} \oplus W| = |\mathcal{F}| + 1$, and $\mathcal{F} \oplus W$ covers the input vertices covered either by $\mathcal{F}$ or by W. □

3.1.2 An efficient algorithm for finding a proper augmenting walk

Algorithm 1 determines if a vertex supports a suitable proper pre-alternating walk W with respect to $\mathcal{F}$, and compute $\mathcal{F} \oplus W$ if one is found. Using it, we may find proper augmenting walks for $\mathcal{F}$ by performing a depth-first search along proper alternating walks W for $\mathcal{F}$ in an attempt to find one which ends in O.

Theorem 18. *Let (G, I, O) be a geometry with $|I| = |O| = k$, $\mathcal{F}$ a proper collection of fewer than k vertex-disjoint paths from I to O, `iter` a positive integer, $i \in I$ a vertex not covered by $\mathcal{F}$, and `visited` $: V(G) \longrightarrow \mathbb{N}$ with `visited`$(x) <$ `iter` for all $x \in V(G)$. Then `AugmentSearch` halts on input $(G, I, O, \mathcal{F}, \texttt{iter}, \texttt{visited}, i)$. Furthermore, let $(\overline{\mathcal{F}}, \overline{\texttt{visited}}, \texttt{status}) = \texttt{AugmentSearch}(G, I, O, \mathcal{F}, \texttt{iter}, \texttt{visited}, i)$.*

- (i). *If `status` = `fail`, then there are no proper augmenting walks for $\mathcal{F}$ starting at i;*
- (ii). *If `status` = `success`, then $\overline{\mathcal{F}}$ is a proper family of vertex-disjoint I – O paths of size $|\mathcal{F}| + 1$ which covers i and all input vertices covered by $\mathcal{F}$, and $\overline{\texttt{visited}}(x) \leqslant \texttt{iter}$ for all $x \in V(G)$.*

Proof — Let G, I, O, $\mathcal{F}$, and `iter` be fixed as above. Throughout the proof, we will consider chains of recursive calls to `AugmentSearch`. One invocation of `AugmentSearch` is the *daughter* of a second invocation if

Algorithm 1 : `AugmentSearch`$(G, I, O, \mathcal{F},$ `iter`, `visited`, $v)$ — searches for an output vertex along pre-alternating walks for $\mathcal{F}$ starting at v, subject to limitations on the end-points of the search paths.

Require: (G, I, O) is a geometry.
Require: $\mathcal{F}$ is a specification for a vertex-disjoint family of I–O paths.
Require: `iter` is a positive integer.
Require: `visited` is an array $V(G) \longrightarrow \mathbb{N}$.
Require: $v \in V(G)$.

```
visited(v) ← iter;
if v ∈ O then return(F, visited, success).
if v ∈ V(F) and v ∉ I and visited(prev(F, v)) < iter then
  (F, visited, status) ← AugmentSearch(G, I, O, F, iter, visited, prev(F, v));
  if status = success then
    F ← RemoveArc(F, prev(F, v) → v);
    return (F, visited, success).
  end if
end if
for all w ∼ v do
  if visited(w) < iter and w ∉ I and (v → w) ∉ A(F) then
    if w ∉ V(F) then
      (F, visited, status) ← AugmentSearch(G, I, O, F, iter, visited, w);
      if status = success then
        F ← AddArc(F, v → w);
        return (F, visited, success).
      end if
    else if visited(prev(F, w)) < iter then
      (F, visited, status) ← AugmentSearch(G, I, O, F, iter, visited, prev(F, w));
      if status = success then
        F ← RemoveArc(F, prev(F, w) → w);
        F ← AddArc(F, v → w);
        return (F, visited, success).
      end if
    end if
  end if
end for
return (F, visited, fail).
```

the first invocation was performed as a step of the second invocation; if one invocation is related to a second invocation by a sequence of daughter-relationships, we will call the second invocation a *descendant* of the first.

At any stage in a particular invocation of `AugmentSearch`, we will refer to the ordered pair (`visited`, v) as the *data pair* of the invocation, where v is the final parameter of the input, and `visited` the second last parameter, including any changes which have been made to it during the invocation. (Though the input parameters of `AugmentSearch` include G, I, O, $\mathcal{F}$, and `iter`, we will occasionally refer to data pairs as the input of an invocation of `AugmentSearch`.) When an invocation of `AugmentSearch` has a data pair (`visited`, v) and makes a daughter invocation, we may describe that invocation as being "daughter invocation for (`visited`, v)" ; similarly, a daughter invocation for (`visited`, v) or the descendant of one is a "descendant invocation for (`visited`, v)".

We define a *probe walk* $\mathcal{W}$ for an ordered pair (`visited`, v) to be a proper pre-alternating walk starting at v such that, for all vertices x in the walk, `visited`$(x) =$ `iter` only if x is at the beginning of $\mathcal{W}$ and $x \in O$ only if x is covered by $\mathcal{F}$ or x is at the end of $\mathcal{W}$. Then, we let $R($`visited`$, v)$ be the set of vertices $x \in V(G)$ which end-points of probe walks for (`visited`, v). We will reduce the problem of determining whether there is a proper augmenting path for $\mathcal{F}$ passing through v to a question of the existence of whether there is an output vertex in

$R(\texttt{visited}, v)$, for visited restricted in a manner described below.

A *canonical walk* for a data pair $(\texttt{visited}, v)$ is a proper alternating walk W with respect to $\mathcal{F}$, such that the following all hold:

(i). $\texttt{visited}(x) \leqslant \texttt{iter}$ for all $x \in V(G)$.

(ii). v is the end-point of W .

(iii). for all vertices x on W , if $\texttt{visited}(x) < \texttt{iter}$, then either $x = v$, or x occurs exactly once in W and is an entry point of W into $\mathcal{F}$.

(iv). for any path P of $\mathcal{F}$, and $x \in V(P)$ which is not in W , $\texttt{visited}(x) < \texttt{iter}$ if and only if either **(a)** there is no entry point of W after x on the path P , or **(b)** there is exactly one entry point p of W after x on the path P , and v lies on P strictly between x and p .

A data pair $(\texttt{visited}, v)$ is itself *canonical* if it has a canonical walk. We will be interested in the behaviour of AugmentSearch on canonical inputs. (Note that the input described in the statement of the Theorem is a special case.) We will show that AugmentSearch essentially performs a depth-first traversal of $R(\texttt{visited}, v)$ along probe walks for $(\texttt{visited}, v)$ in an attempt to find an output vertex. If it succeeds, it has traversed a proper augmenting walk $\overline{\mathtt{W}}$ for $\mathcal{F}$, and can construct $\mathcal{F} \oplus \overline{\mathtt{W}}$.

Suppose W is a canonical walk for a data pair $(\texttt{visited}, v)$. It is easy to show that if we extend W to a longer walk $W' = Wvw_1w_2 \cdots w_N$ for some $N \geqslant 1$, and W' is a canonical walk for some data pair $(\texttt{visited}^*, w_N)$, then W is a canonical walk for $(\texttt{visited}^*, v)$. We will use this fact frequently in the two Lemmas below.

Lemma 18-1. *Suppose that* $(\texttt{visited}, v)$ *has a canonical walk* W . *If* $R(\texttt{visited}, v)$ *does not contain any output vertices,* AugmentSearch *halts on input data* $(\texttt{visited}, v)$ *, with output value* $(\mathcal{F}, \overline{\texttt{visited}}, \texttt{fail})$ *; where* $\overline{\texttt{visited}}$ *differs from* visited *only in that* $\overline{\texttt{visited}}(x) = \texttt{iter}$ *for all* $x \in R(\texttt{visited}, v)$ *, and where* $(\overline{\texttt{visited}}, v)$ *also has the canonical walk* W .

Proof — We will proceed by induction on the length ℓ of the longest probe walk for $(\texttt{visited}, v)$. Regardless of the value of ℓ , line 1 transforms the data pair $(\texttt{visited}, v)$ to $(\texttt{visited}^{(1)}, v)$, where $\texttt{visited}^{(1)}$ differs from visited in that $\texttt{visited}^{(1)}(v) = \texttt{iter}$; then, any canonical walk for $(\texttt{visited}, v)$ is also a canonical walk for $(\texttt{visited}^{(1)}, v)$. As well, it cannot be that $v \in O$: then the condition on line 2 will not be satisfied.

If $\ell = 0$, the condition on lines 3 cannot be satisfied, and the condition of line 11 is not satisfied by any neighbor $w \sim v$. Then, line 28 will ultimately be executed, returning $(\mathcal{F}, \texttt{visited}^{(1)}, \texttt{fail})$. Because $R(\texttt{visited}, v) = \{v\}$, the proposition holds in this case.

Otherwise, suppose $\ell > 0$, and that the proposition holds for canonical data pairs whose probe walks all have length less than ℓ . Consider the vertices which may be the subject of a daughter invocation of AugmentSearch:

1. If $v \notin I$ and v is covered by $\mathcal{F}$, and z is the predecessor of v in the paths of $\mathcal{F}$, then $(\texttt{visited}, v)$ has probe walks starting with the arc $v \to z$ if and only if $\texttt{visited}^{(1)}(z) = \texttt{visited}(z) < \texttt{iter}$. If this holds, then a daughter invocation of AugmentSearch with input data $(\texttt{visited}^{(1)}, z)$ is performed.

   In this case, note that $(\texttt{visited}^{(1)}, z)$ has probe walks ending in O only if $(\texttt{visited}, v)$ does; then $R(\texttt{visited}^{(1)}, z)$ is disjoint from O , and all of the probe walks of $(\texttt{visited}^{(1)}, z)$ are strictly shorter than those of $(\texttt{visited}, v)$. Let $W^{(1)} = Wvz$: because W is a canonical walk, $z \in V(W)$ only if z occurs only once in W and is an entry point of W into $\mathcal{F}$, in which case the edge vz is never traversed by W . Then, it is easy to show that $W^{(1)}$ is a canonical walk for $(\texttt{visited}^{(1)}, z)$. By the inductive hypothesis, AugmentSearch will halt on input $(\texttt{visited}^{(1)}, z)$ and return a value $(\mathcal{F}, \texttt{visited}^{(2)}, \texttt{fail})$, where $\texttt{visited}^{(2)}$ differs from $\texttt{visited}^{(1)}$ only in that $\texttt{visited}^{(2)}(x) = \texttt{iter}$ for all $x \in R(\texttt{visited}^{(1)}, z) \subseteq R(\texttt{visited}, v)$, and where $W^{(1)}$ is a canonical walk for $(\texttt{visited}^{(2)}, z)$. Then, W is a canonical walk for $(\texttt{visited}^{(2)}, v)$.

   Otherwise, if $\texttt{visited}^{(1)}(z) = \texttt{iter}$, if $v \in I$, or if $\mathcal{F}$ does not cover v , let $\texttt{visited}^{(2)} = \texttt{visited}^{(1)}$; W is a canonical walk for $(\texttt{visited}^{(2)}, v)$ in this case as well.

2. Suppose that at some iteration of the for loop starting at line 10, the data of AugmentSearch is a data pair $(\texttt{visited}^{(h)}, v)$ for which W is a canonical walk, $\texttt{visited}^{(h)}(v) = \texttt{iter}$, and w is a neighbor of v

satisfying the conditions of lines 11 and 12. Note that $(\texttt{visited}^{(h)}, w)$ has probe walks ending in O only if $(\texttt{visited}, v)$ does; then, $R(\texttt{visited}^{(h)}, w)$ is disjoint from O, and all of the probe walks of the former are strictly shorter than those of the latter. Let $W^{(h)} = Wvw$; because w is not covered by $\mathcal{F}$, the fact that $\texttt{visited}^{(h)}(w) = \texttt{visited}(w) < \texttt{iter}$ implies that w does not occur in W. Because $w \notin I$, we know that $W^{(h)}$ is a proper alternating walk. In particular, $W^{(h)}$ is a canonical walk for $(\texttt{visited}^{(h)}, w)$.

By the inductive hypothesis, `AugmentSearch` will then halt on input $(\texttt{visited}^{(h)}, w)$ and return a value $(\mathcal{F}, \texttt{visited}^{(h+1)}, \texttt{fail})$, where $\texttt{visited}^{(h+1)}$ differs from $\texttt{visited}^{(h)}$ only in that $\texttt{visited}^{(h+1)}(x) = \texttt{iter}$ for all $x \in R(\texttt{visited}^{(h)}, w) \subseteq R(\texttt{visited}, v)$, and where $W^{(h)}$ is a canonical walk for $(\texttt{visited}^{(h+1)}, w)$. Then, W is a canonical walk for $(\texttt{visited}^{(h+1)}, v)$.

3. Suppose that at some iteration of the for loop starting at line 10, the data of `AugmentSearch` is a data pair $(\texttt{visited}^{(h)}, v)$ for which W is a canonical walk, $\texttt{visited}^{(h)}(v) = \texttt{iter}$, and w is a neighbor of v satisfying the conditions of lines 11 and 18. Then, w is covered by a path $P \in \mathcal{F}$ and has a well-defined predecessor z in P. Let $W^{(h)} = Wvwz$: this is an alternating walk with respect to $\mathcal{F}$.

The walk $W^{(h)}$ is monotonic only if w is further from the initial point of the path $P \in \mathcal{F}$ than any entry point of W on P. If P contains no entry points of W into $\mathcal{F}$, this is satisfied. Otherwise, let y be the final entry point of W into P.

- Suppose that v is not covered by P. Because $(\texttt{visited}^{(h)}, v)$ is canonical, every vertex x on the path P from the initial point up to (but possibly not including) y has $\texttt{visited}^{(h)}(x) = \texttt{iter}$. Because $\texttt{visited}^{(h)}(z) < \texttt{iter}$, z is at least as far along P as y is; then, w is strictly further. Thus, $W^{(h)}$ is monotonic.
- If v is covered by P, then every vertex x on P with $\texttt{visited}(x) < \texttt{iter}$ either is at least as far as y on P, or has the property that y is the only entry point between x and the end of P, and that v lies between x and y. However, if there are more than zero vertices of the second type, then v has a predecessor z in P with $\texttt{visited}(z) < \texttt{iter}$. Then, from the analysis of part *1* above, all vertices x which precede v in P with $\texttt{visited}(x) < \texttt{iter}$ are in $R(\texttt{visited}^{(1)}, z)$, and thus have $\texttt{visited}^{(h)}(x) = \texttt{visited}^{(2)}(x) = \texttt{iter}$. Then, $W^{(h)}$ is monotonic if and only if w is further along P than y, which reduces to the analysis of the preceding case.

Because $\texttt{visited}^{(h)}(z) < \texttt{iter}$, either z is not in W, or it occurs exactly once as an entry point of W into $\mathcal{F}$. Because $\texttt{visited}^{(h)}(w) < \texttt{iter}$ and w is further along P than any entry point of W, w does not occur in W at all. Then, neither vw nor wz are traversed by W, in which case $W^{(h)}$ is a proper alternating walk. In particular, it is a canonical walk for $(\texttt{visited}^{(h)}, z)$.

Again, $(\texttt{visited}^{(h)}, z)$ has probe walks ending in O only if $(\texttt{visited}, v)$ does; then, $R(\texttt{visited}^{(h)}, z)$ is disjoint from O, and all of the probe walks of the former are strictly shorter than those of the latter. By the inductive hypothesis, `AugmentSearch` will then halt on input $(\texttt{visited}^{(h)}, z)$ and return a value $(\mathcal{F}, \texttt{visited}^{(h+1)}, \texttt{fail})$, where $\texttt{visited}^{(h+1)}$ differs from $\texttt{visited}^{(h)}$ only in that $\texttt{visited}^{(h+1)}(x) = \texttt{iter}$ for all $x \in R(\texttt{visited}^{(h)}, z) \subseteq R(\texttt{visited}, v)$, and where $W^{(h)}$ is a canonical walk for $(\texttt{visited}^{(h+1)}, z)$. Then, W is a canonical walk for $(\texttt{visited}^{(h+1)}, v)$.

By induction on the number of neighbors $w \sim v$ satisfying the conditions of lines 11, 12, and 18, the data $(\overline{\texttt{visited}}, v)$ when the **for** loop terminates and line 28 is executed will be a nearly canonical pair, and $\overline{\texttt{visited}}$ differs from $\texttt{visited}$ only on elements of $R(\texttt{visited}, v)$.

It remains to show that $\overline{\texttt{visited}}(x) = \texttt{iter}$ for all $x \in R(\texttt{visited}, v)$. We have shown this already for $x = v$; then, let $r \in R(\texttt{visited}, v) \setminus \{v\}$. By definition there is a probe walk $\mathcal{W}$ for $(\texttt{visited}, v)$ ending in r. The vertex w immediately following v on $\mathcal{W}$ will be either tested on line 3 or line 11 as a neighbor of v; then, there exists indices h' such that $\mathcal{W}$ is not a probe walk of $(\texttt{visited}^{(h')}, v)$. Let $h > 0$ be the largest integer such that $\mathcal{W}$ *is* a probe walk for $(\texttt{visited}^{(h)}, v)$: then, there are vertices $x \neq v$ in $\mathcal{W}$ such that $\texttt{visited}^{(h+1)}(x) = \texttt{iter}$. Let $y \in R$ be the last such vertex in $\mathcal{W}$, let $\mathcal{W}'$ be the segment of $\mathcal{W}$ from y onwards: then $r \in R(\texttt{visited}^{(h+1)}, y)$. Because $\texttt{visited}^{(h+1)}(y) = \texttt{iter}$, there must have been a descendant invocation for $(\texttt{visited}^{(h)}, v)$ which had input data $(\texttt{visited}^{*}, y)$ for some function $\texttt{visited}^{*}$: it is not difficult to show that $\texttt{visited}^{*}(y) < \texttt{iter}$. By

induction on daughter invocations using the analysis above, we may show that $(\texttt{visited}^*, y)$ is a canonical data pair with probe walks strictly shorter than ℓ: then, for all $x \in R(\texttt{visited}^*, y)$, we have $\texttt{visited}^{(h+1)}(x) = \texttt{iter}$. However, because $\texttt{visited}^{(h+1)}(x) < \texttt{iter}$ implies $\texttt{visited}^*(x) < \texttt{iter}$, and because all vertices $x \in V(\mathcal{W})$ after y have $\texttt{visited}^{(h+1)}(x) < \texttt{iter}$, $\mathcal{W}'$ is a probe walk for $(\texttt{visited}^*, y)$. Then, we have $\texttt{visited}^{(h+1)}(r) = \texttt{iter}$.

Because $\overline{\texttt{visited}}(x) = \texttt{iter}$ if and only if $\texttt{visited}^{(h)}(x) = \texttt{iter}$ for some $h \geqslant 1$, we then have $\overline{\texttt{visited}}(r) = \texttt{iter}$ for any $r \in R(\texttt{visited}, v)$. By induction, the Lemma then follows. ◊

Lemma 18-2. *Suppose that* $(\texttt{visited}, v)$ *is a canonical data pair. If* $R(\texttt{visited}, v)$ *contains an output vertex,* `AugmentSearch` *halts on input data* $(\texttt{visited}, v)$, *with output value* $(\mathcal{F} \oplus \overline{\mathcal{W}}, \overline{\texttt{visited}}, \texttt{success})$; *where* $\overline{\mathcal{W}}$ *is a probe walk for* $(\texttt{visited}, v)$ *ending in* O, *and* $\overline{\texttt{visited}}$ *differs from* $\texttt{visited}$ *only in that* $\overline{\texttt{visited}}(x) = \texttt{iter}$ *only for* x *in some subset of* $R(\texttt{visited}, v)$.

Proof — We induct on the length $\ell \in \mathbb{N}$ of the longest probe walk for $(\texttt{visited}, v)$ ending in O. If $\ell = 0$, then $v \in O$, and the result holds trivially. Otherwise, suppose $\ell > 0$ and that the result holds for those canonical data pairs $(\texttt{visited}^*, x)$ which have probe walks of length less than ℓ ending in O.

Let W be a canonical walk for $(\texttt{visited}, v)$. Consider the sequence of vertices $w_1, w_2, \cdots, w_M$ which are tested (either on line 4, line 13, or line 19) in the course of the invocation of `AugmentSearch`. We let $\texttt{visited}^{(1)}$ differ from $\texttt{visited}$ in that $\texttt{visited}^{(1)}(v) = \texttt{iter}$, and from this define $\texttt{visited}^{(j)}$ for $j > 1$ by letting $\texttt{visited}^{(j+1)}$ be the second component of the output of the daughter invocation with input data $(\texttt{visited}^{(j)}, w_j)$. (If the daughter invocation with data pair $(\texttt{visited}^{(M)}, w_M)$ halts, this sequence extends to $\texttt{visited}^{(M+1)}$.)

Let $\mathcal{W}$ be a probe walk for $(\texttt{visited}, v)$ which ends at a vertex $\omega \in O$, and let $1 \leqslant N \leqslant M + 1$ be the largest integer such that $R(\texttt{visited}^{(j)}, w_j)$ is disjoint from O for all $j < N$. If $N = M + 1$, this means that the invocation of `AugmentSearch` on input data $(\texttt{visited}^{(M)}, w_M)$ halted with `fail` in the final part of its' output value, and that there are no neighbors of $w \sim v$ which can satisfy the conditions of lines 11, 12, and 18 (due to the choice of M as the length of the sequence of daughter-invocations). However, we may show by induction that for all $1 \leqslant j \leqslant N$, $\mathcal{W}$ is a probe walk for $(\texttt{visited}^{(j)}, v)$, which is canonical:

- This follows immediately for $j = 1$, because $\texttt{visited}^{(1)}$ only differs from $\texttt{visited}$ at v, and thus has $\mathcal{W}$ as a probe walk and W as a canonical walk.

- Suppose for some $1 \leqslant j < M$ that W is a canonical walk for $(\texttt{visited}^{(j)}, v)$, that $\mathcal{W}$ is a probe walk for $(\texttt{visited}^{(j)}, v)$, and that $\texttt{visited}^{(j)}(v) = \texttt{iter}$. Then we can extend W to a canonical walk $W^{(j)}$ for $(\texttt{visited}^{(j)}, w_j)$: either by setting $W^{(j)} = Wvw_j$ in the case that $(w_j \to v) \in A(\mathcal{F})$ or w_j is not covered by $\mathcal{F}$, or by setting $W^{(j)} = Wvzw_j$ where $(z \to w_j) \in A(\mathcal{F})$ otherwise. Because $R(\texttt{visited}^{(j)}, w_j)$ contains no output vertices, by Lemma 18-1 we know that $\texttt{visited}^{(j+1)}$ differs from $\texttt{visited}^{(j)}$ only on $R(\texttt{visited}^{(j)}, w_j)$ and that $W^{(j)}$ is a canonical walk for $(\texttt{visited}^{(j+1)}, w_j)$. Then, W is a canonical walk for $(\texttt{visited}^{(j+1)}, v)$.

  For any vertex x in $\mathcal{W}$, the sub-path $\mathcal{W}_x$ from x to ω is a probe walk for $(\texttt{visited}^{(j)}, x)$ of length less than ℓ. If $\mathcal{W}$ has a non-trivial intersection with $R(\texttt{visited}^{(j)}, w_j)$, then some vertex $x \in V(\mathcal{W})$ is the first such vertex which is given as part of an input data pair $(\texttt{visited}^*, x)$ for a descendant invocation for $(\texttt{visited}^{(j)}, w_j)$. By induction on the recursion depth from v to x, we may show that there is then a probe walk $\mathcal{W}^*$ for $(\texttt{visited}^{(j)}, v)$ ending at x, and that $W\mathcal{W}^*$ is a canonical walk for $(\texttt{visited}^*, x)$; and precisely because x is the first vertex of $\mathcal{W}$ which is visited in a descendant invocation for $(\texttt{visited}^{(j)}, w_j)$, we know that $\texttt{visited}^*(y) < \texttt{iter}$ for all $y \in V(\mathcal{W}) \smallsetminus \{v, x\}$. Then, $\mathcal{W}_x$ is a probe walk for $(\texttt{visited}^*, x)$, and by the induction hypothesis, this invocation of `AugmentSearch` then terminates with `success` as the last part of its' output. Again by induction on the recursion depth, we may also show that the invocation of `AugmentSearch` with data $(\texttt{visited}^{(j)}, w_j)$ would also terminate with `success` as the last part of its' output. But because $j < N$, this cannot happen by Lemma 18-1 — from which it follows that $\mathcal{W}$ is disjoint from $R(\texttt{visited}^{(j)}, w_j)$. Thus $\mathcal{W}$ is also a probe walk for $(\texttt{visited}^{(j+1)}, v)$.

By induction, $\mathcal{W}$ is a probe walk for $(\texttt{visited}^{(N)}, v)$, so it must be that $N \leqslant M$. By the choice of N, there is then a probe walk $\mathcal{W}'$ for $(\texttt{visited}^{(N)}, w_N)$ which ends in O.

Because w_N is part of the input to a daughter invocation of `AugmentSearch`, we have $\texttt{visited}^{(N)}(w_N) < \texttt{iter}$; thus we can easily extend $\mathcal{W}'$ (by one or two vertices, depending on whether w_N is a neighbor of v or the predecessor in $\mathcal{F}$ of a neighbor of v) to form a probe walk $\mathcal{W}''$ for $(\texttt{visited}^{(N)}, v)$. Because $\mathcal{W}''$ will also be a probe walk for $(\texttt{visited}, v)$, it has length at most ℓ; then $\mathcal{W}'$ is strictly shorter than ℓ in length. By the induction hypothesis, the invocation of `AugmentSearch` on input data $(\texttt{visited}^{(N)}, w_N)$ then halts, and returns the output value $(\mathcal{F} \oplus \overline{\mathcal{W}}', \overline{\texttt{visited}}, \texttt{success})$, where $\overline{\mathcal{W}}'$ is a probe walk for $(\texttt{visited}^{(N)}, w_N)$ ending in O, and where $\overline{\texttt{visited}}$ differs from $\texttt{visited}^{(N)}$ only on a subset of $R(\texttt{visited}^{(N)}, w_N)$. We proceed by cases:

- If v is covered by a path of $\mathcal{F}$, $v \notin I$, and w_N is the predecessor of v in $\mathcal{F}$, then $\overline{\mathcal{W}} = v w_N \overline{\mathcal{W}}'$ is a probe walk for $(\texttt{visited}, v)$ ending in O. Note that $A(\mathcal{F} \oplus \overline{\mathcal{W}}) = A(\mathcal{F} \oplus \overline{\mathcal{W}}') \smallsetminus \{w_N \to v\}$; then, the value which is returned as output on line 7 is $(\mathcal{F} \oplus \overline{\mathcal{W}}, \overline{\texttt{visited}}, \texttt{success})$.

- If w_N is not covered by a path of $\mathcal{F}$, then $w_N \sim v$, and the walk $\overline{\mathcal{W}} = v w_N \overline{\mathcal{W}}'$ is a probe walk for $(\texttt{visited}, v)$ ending in O. Note that $A(\mathcal{F} \oplus \overline{\mathcal{W}}) = A(\mathcal{F} \oplus \overline{\mathcal{W}}') \cup \{v \to w_N\}$; then, the value which is returned as output on line 16 is $(\mathcal{F} \oplus \overline{\mathcal{W}}, \overline{\texttt{visited}}, \texttt{success})$.

- If neither of the previous two cases apply, it must be that w_N is the predecessor in $\mathcal{F}$ of some third vertex $u \sim v$. Because w_N is part of the input to a daughter invocation for $(\texttt{visited}^{(N)}, v)$, we know that $\texttt{visited}^{(N)}(u) < \texttt{iter}$: then, $\overline{\mathcal{W}} = v u w_N \overline{\mathcal{W}}'$ is a probe walk for $(\texttt{visited}, v)$ ending in O. Note that $A(\mathcal{F} \oplus \overline{\mathcal{W}}) = \big[A(\mathcal{F} \oplus \overline{\mathcal{W}}) \smallsetminus \{w_N \to u\}\big] \cup \{v \to u\}$; then, the value which is returned as output on line 23 is $(\mathcal{F} \oplus \overline{\mathcal{W}}, \overline{\texttt{visited}}, \texttt{success})$.

Finally, because $R(\texttt{visited}^{(N)}, w_N) \subseteq R(\texttt{visited}, v)$, and because $\texttt{visited}^{(N)}$ differs from $\texttt{visited}$ only on $R(\texttt{visited}(i), w_i) \subseteq R(\texttt{visited}, v)$ for $1 \leqslant i < N$, it follows that $\overline{\texttt{visited}}$ differs from $\texttt{visited}$ only on a subset of $R(\texttt{visited}, v)$, with $\overline{\texttt{visited}}(x) = \texttt{iter}$ on that subset. Thus, if the Lemma holds for pairs $(\texttt{visited}, v)$ having probe walks of length less than $\ell \geqslant 0$ ending in O, it also holds for such pairs with probe walks ending in O of length $\ell + 1$. By induction, the Lemma then holds. ♢

To prove the Theorem, it then suffices to note that for a function $\texttt{visited} : V(G) \longrightarrow \mathbb{N}$ with $\texttt{visited}(x) < \texttt{iter}$ for all $x \in V(G)$, probe walks for $(\texttt{visited}, i)$ are just proper alternating walks with respect to $\mathcal{F}$ which start at i, in which case such a probe walk $\overline{\mathcal{W}}$ which ends in O is a proper augmenting walk for $\mathcal{F}$. Then all the various parts of the Theorem follow from Lemmas 18-1 and 18-2 collectively. □

Run-time analysis. Because `AugmentSearch` marks each vertex v with $\texttt{visited}(v) \leftarrow \texttt{iter}$ when it visits v, each vertex is only visited once. At each vertex, each of the neighbors $w \sim v$ are tested for if they fulfill the condition of line 3, or of lines 11, 12, and 18. Because computing `prev`, `AddArc`, and `RemoveArc` is constant-time for $\mathcal{F}$ a collection of vertex-disjoint paths (or differing only slightly from one as described in the discussion on implementation details), the amount of work in an invocation to `AugmentSearch` for a vertex $v \in V(G)$ is $O(\deg v)$, neglecting the work performed in descendant invocations. Summing over all vertices $v \in V(G)$, the run-time of `AugmentSearch` is then $O(m)$ for an input as described in the statement of Theorem 18.

3.1.3 An efficient algorithm for constructing a path cover for (G, I, O)

Using `AugmentSearch` as a subroutine to build successively larger proper families of vertex-disjoint $I - O$ paths, Algorithm 2 describes a straightforward subroutine which attempts to build a path cover for (G, I, O).

Corollary 19. *Let (G, I, O) be a geometry with $|I| = |O|$: then* `BuildPathCover` *halts on input (G, I, O). Furthermore, let $\sigma = \texttt{BuildPathCover}(G, I, O)$. If $\sigma = \texttt{fail}$, then (G, I, O) does not have a causal flow; otherwise, σ is a path cover $\mathcal{F}$ for (G, I, O).*

Proof — Suppose (G, I, O) has a causal flow: then it has a collection of $k = |I| = |O|$ vertex-disjoint $I - O$ paths by Lemma 3. Then, by Corollary 15, for any proper collection $\mathcal{F}$ of vertex-disjoint $I - O$ paths with

Algorithm 2: `BuildPathCover`(G, I, O) — tries to build path cover for (G, I, O)

Require: (G, I, O) is a geometry.

1: **let** $\mathcal{F}$: an empty collection of vertex-disjoint dipaths in G
2: **let** `visited` $: V(G) \longrightarrow \mathbb{N}$ be an array initially set to zero
3: **let** `iter` $\leftarrow 0$
4: **for all** $i \in I$ **do**
5:   `iter` $\leftarrow$ `iter` $+ 1$
6:   $(\mathcal{F}, \texttt{visited}, \texttt{status}) \leftarrow \texttt{AugmentSearch}(G, I, O, \mathcal{F}, \texttt{iter}, \texttt{visited}, i)$
7:   **if** `status` $=$ `fail` **then return** `fail`
8: **end for**
9: **if** $V(G) \setminus V(\mathcal{F}) = \varnothing$ **then**
10:   **return** $\mathcal{F}$
11: **else**
12:   **return** `fail`
13: **end if**

$|\mathcal{F}| < k$, there is a proper augmenting walk for $\mathcal{F}$ starting at any $i \in I$ which is not covered by $\mathcal{F}$. For such a collection $\mathcal{F}$ and vertex i, if $\texttt{visited}(x) < \texttt{iter}$ for all $x \in V(G)$, $\texttt{AugmentSearch}(G, I, O, \mathcal{F}, \texttt{iter}, \texttt{visited}, i)$ returns $(\mathcal{F} \oplus W, \overline{\texttt{visited}}, \texttt{success})$, where $\overline{\texttt{visited}}(x) \leqslant \texttt{iter}$ for all $x \in V(G)$, and where W is a proper augmenting walk for $\mathcal{F}$ starting at i. Then, $\mathcal{F} \oplus W$ is a proper collection of vertex-disjoint paths, covering i and the input vertices covered by $\mathcal{F}$, and with $|\mathcal{F} \oplus W| = |\mathcal{F}| + 1$. By induction, we may then show that at the end of the **for** loop starting at line 4, $\mathcal{F}$ will be a family of vertex-disjoint $I - O$ paths which covers all of I, in which case $|\mathcal{F}| = k$. If all of the vertices of $V(G)$ are covered by $\mathcal{F}$, $\mathcal{F}$ is then a path cover for (G, I, O), and `BuildPathCover` returns $\mathcal{F}$. Taking the contrapositive, if `BuildPathCover`(G, I, O) returns `fail`, then (G, I, O) has no path cover.

Conversely, if `BuildPathCover`(G, I, O) returns `fail`, then either the condition of line 7 failed, or the condition of line 12 failed. If the former is true, then by Theorem 18 there were no proper augmenting walks for some proper collection $\mathcal{F}$ of fewer than k disjoint I–O paths, in which case by Corollary 15 there is no such collection of size k, and thus no causal path cover for (G, I, O). Otherwise, $\mathcal{F}$ is a maximum-size collection of disjoint paths from I to O, but is not a path cover for (G, I, O); then by Theorem 9, there again is no causal path cover for (G, I, O). In either case, there is no causal flow for (G, I, O) by Theorem 8. The result then holds. ☐

Run-time analysis. `BuildPathCover` iterates through $k = |I|$ input vertices as it increases the size of the collection of vertex-disjoint paths, invoking `AugmentSearch` for each one. The running time for this portion of the algorithm is then $O(km)$. As this is larger than the time required to initialize `visited` or to determine if there is an element $v \in V(G)$ such that $v \notin V(\mathcal{F})$, this dominates the asymptotic running time of `BuildPathCover`.

3.2 Efficiently finding a causal order for a given successor function

Given a path cover $\mathcal{C}$ for a geometry (G, I, O), and in particular the successor function f of $\mathcal{C}$, we are interested in determining if the natural pre-order $\preccurlyeq$ for f is a partial order, and constructing it if so. In this section, I present an efficient algorithm to determine whether or not $\preccurlyeq$ is a partial order, by reduction to the transitive closure problem on digraphs.

3.2.1 The Transitive Closure Problem

Any binary relation R can be regarded as defining a digraph D with $(x \to y) \in A(D) \iff (x \mathsf{R} y)$. Chains of related elements can then be described by directed walks in the digraph D. This motivates the following definition:

Definition 20. Let f be a successor function for a geometry (G, I, O) : the *influencing digraph* $\mathcal{I}_f$ is then the directed graph with vertices $V(\mathcal{I}_f) = V(G)$, where $(x \to y) \in A(\mathcal{I}_f)$ if one of $y = x$, $y = f(x)$, or $y \sim f(x)$ hold.

The three types of arcs in Definition 20 correspond to the relations in Equation 2, whose transitive closure is the natural pre-order. Note that aside from self-loops $x \to x$, the arcs in $\mathcal{I}_f$ correspond directly to the two varieties of segments of influencing walks. (This is an alternative way of proving Lemma 6.)

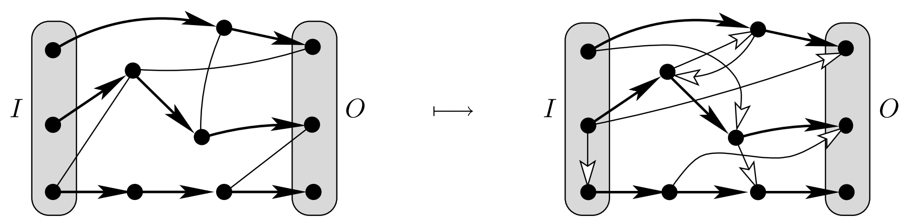


Figure 6: On the left: a geometry (G, I, O) with a path cover $\mathcal{C}$. Arrows represent the action of the successor function $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$ of $\mathcal{C}$. On the right: the corresponding influencing digraph $\mathcal{I}_f$. Solid arrows represent arcs of the form $x \to f(x)$, and hollow arrows represent arcs $x \to y$ for $y \sim f(x)$. (Self-loops are omitted for clarity.)

It is natural to also speak of transitive closures of binary relations in graph-theoretic terms, as follows:

Definition 21. For a digraph D , the *transitive closure* of D is the digraph T with $V(T) = V(D)$, and such that $(x \to y) \in A(T)$ if and only if there is a non-trivial[6] directed walk from x to y in the digraph D .

Thus, $x \to y$ is an arc in the transitive closure of $\mathcal{I}_f$ and only if $x \preccurlyeq y$, or equivalently iff there is an influencing walk for $\mathcal{C}$ from x to y in G.

Transitive Closure Problem. *Given a digraph D , determine it's transitive closure T .*

The Transitive Closure problem is known to be efficiently solvable. Algorithm 3 presents on solution, which is (a paraphrasing of) the pseudocode of Figure 3.8 from [13]. This algorithm is a simple modification of Tarjan's algorithm for finding *strongly connected components* of digraphs (equivalence classes of mutually reachable vertices using directed walks), which finds the transitive closure by determining the "descendants" of each $x \in V(D)$:

$$\texttt{Desc}(x) \;=\; \Big\{\, y \in V(D) \;\Big|\; D \text{ contains a non-trivial directed walk from } x \text{ to } y \Big\} \quad . \tag{3}$$

The following is an overview of Algorithm 3: interested readers may refer to [13] for a more complete analysis.

- A di-connected component of D is an equivalence class of vertices which can be reached from each other by non-trivial directed walks in D . Tarjan's algorithm detects these components by performing a depth-first search which traverses arcs of D , and detecting when it has traversed a directed cycle in D .
- A stack is used to keep track of vertices of the digraph have been visited, but whose di-connected component has not yet been completely determined. When the vertices belonging to a a given component are determined, we pop them off of the stack (line 14) and insert them into a set representing that component.
- We say that v precedes w in the ordering of the stack if v is on the stack and w is not, or if v is lower on the stack than w is. Then, we may keep track of the "root" $\texttt{Root}(v)$ of v , which is an upper bound on the stack-minimal vertex of the component containing v. At first, we set the root of v to itself, and we always ensure that $\texttt{Root}(v) \leqslant v$.

  Suppose we discover a descendant w of v such that $\texttt{Root}(w) \leqslant \texttt{Root}(v) \leqslant v$. Then v is a descendant of $\texttt{Root}(w)$, which is in a common component with w by definition. Because w is also a descendent of v , v

[6]Note that if a vertex x has a loop $x \to x$ (which are permitted in digraphs), then the directed walk $x \to x$ is a non-trivial walk.

Algorithm 3: Figure 3.8 of [13] — an algorithm for transitive closure of a digraph

1: **procedure** `SimpleTC`(v)
2: **begin**
3:   `Root`$(v) \leftarrow v$; `Comp`$(v) \leftarrow$ `nil`
4:   `PUSH`$(v,$ `stack`$)$
5:   `Desc`$(v) \leftarrow \{ w \in V(D) \mid (v \to w) \in A(D) \}$
6:   **for all** w **such that** $(v \to w) \in A(D)$ **do**
7:     **if** (w is not already visited) **then** `SimpleTC`(w)
8:     **if** `Comp`$(w) =$ `nil` **then** `Root`$(v) \leftarrow \min($`Root`$(v),$ `Root`$(w))$
9:     `Desc`$(v) \leftarrow$ `Desc`$(v) \cup$ `Desc`(w)
10:   **end for**
11:   **if** `Root`$(v) = v$ **then**
12:     create a new component C
13:     **repeat**
14:       **let** $w \leftarrow$ `POP`(`stack`)
15:       `Comp`$(w) \leftarrow C$
16:       insert w into the component C
17:       `Desc`$(w) \leftarrow$ `Desc`(v)
18:     **until** $w = v$
19:   **end if**
20: **end**

21: **procedure** `main`
22: **begin**
23:   **let** `stack` $\leftarrow \varnothing$
24:   **for all** $v \in V(D)$ **do**
25:     **if** (v is not already visited) **then** `SimpleTC`(v)
26:   **end for**
27: **end**

must be in a common component with w. Then `Root`(w) is the smallest known vertex in that component: we update `Root`$(v) \leftarrow$ `Root`(w) to improve the known minimum for v.

- Because vertices are only allocated to a di-connected component after they are popped off the stack, we may test each of the descendants w of v to see if they have been allocated to a component, rather than testing if `Root`$(w) \leqslant$ `Root`(v). If not, then v is in a common component with w, and we update `Root`(v) to be the minimum of `Root`(v) and `Root`(w) on line 8, as in the previous case.

- If `Root`$(v) = v$ on line 11, then v is the stack-minimal element of its' component: then any vertices higher than v on the stack will be in the same component as v. Conversely, because all descendants of v have been visited by that point, all of the vertices in the same component as v are still on the stack. Thus, we may pop them off the stack and allocate them to a component, until we have removed v off of the stack (lines 11 through 19).

- As we determine the connected components of the digraph, we may maintain the sets of descendants of each vertex: if $(v \to w) \in A(D)$, then the descendants of w are all also descendants of v, so we ensure that `Desc`$(w) \subseteq$ `Desc`(v) (as on line 9).

The above is performed for all vertices $v \in V(G)$ to obtain the transitive closure.

Algorithm 3 is sufficient to build the natural pre-order $\preccurlyeq$ for a successor function f. However, the output does not indicate whether $\preccurlyeq$ is a partial order, and it performs work that is unnecessary if $\preccurlyeq$ is not actually a partial order. We may also take advantage of the availability of the path cover $\mathcal{C}$ which is given as input, which is not available in the more general Transitive Closure problem. Therefore, we are interested in adapting Algorithm 3 to the application of finding a causal order.

3.2.2 Chain decompositions with respect to the path cover $\mathcal{C}$

Let $\mathcal{C}$ be a path cover for (G, I, O) with successor function f. The transitive closure of the influencing digraph $\mathcal{I}_f$ will often have high maximum degree: because the longest path in $\mathcal{C}$ has at least n/k vertices, and the end-point of this path will be at the terminus of arcs coming from every vertex on the path, the maximum in-degree of the transitive closure is at least n/k; and similarly for the maximum out-degree. In Algorithm 3, this implies that the set $\texttt{Desc}(v)$ may become comparable to $V(G)$ in size. In order to construct the arc-lists of the transitive closure reasonably efficiently, we want to reduce the effort required in determining the sets $\texttt{Desc}(v)$.

A standard approach to this problem would be to find a *chain decomposition* [13] for $\mathcal{I}_f$, which is a collection of vertex-disjoint dipaths of $\mathcal{I}_f$ which cover all of $\mathcal{I}_f$. By the definition of the influencing digraph, $\mathcal{C}$ itself is such decomposition of $\mathcal{I}_f$. Then, using a chain decomposition with respect to $\mathcal{C}$, we can efficiently represent $\texttt{Desc}(x)$ in terms of the first vertex y in each path of $\mathcal{C}$ such that $y \in \texttt{Desc}(x)$.

Definition 22. Let $\mathcal{C} = \{\mathcal{P}_j\}_{j\in K}$ be a parameterization of the paths of a path cover $\mathcal{C}$ for a geometry (G, I, O), let f be the successor function of $\mathcal{C}$, and let $\preccurlyeq$ be the natural pre-order for f. Then, for $x \in V(G)$ and $j \in K$, the *supremum* $\sup_j(x)$ *of* x *in* $\mathcal{P}_j$ is the minimum integer $m \in \mathbb{N}$, such that $x \preccurlyeq y$ for all vertices $y \in V(\mathcal{P}_j)$ which are further than distance m from the initial vertex of $\mathcal{P}_j$.

We may use the suprema of x in the paths of $\mathcal{C}$ to characterize the natural pre-order for f:

Lemma 23. *Let $\mathcal{C} = \{\mathcal{P}_j\}_{j\in K}$ be a parameterization of the paths of a path cover $\mathcal{C}$ for a geometry (G, I, O), let f be the successor function of $\mathcal{C}$, let $\preccurlyeq$ be the natural pre-order for f, and let $L : V(G) \longrightarrow \mathbb{N}$ map vertices $x \in V(G)$ to the distance of x from the initial point of the path of $\mathcal{C}$ which contains x. Then*

$$x \preccurlyeq y \quad\iff\quad \sup_j(x) \leqslant L(y) \tag{4}$$

holds for all $x \in V(G)$ and $y \in V(\mathcal{P}_j)$, for any $j \in K$.

Proof — Let $x \in V(G)$, and fix $\mathcal{P}_j \in \mathcal{C}$. Let $v \in V(\mathcal{P}_j)$ be such that $L(v) = \sup_j(x)$. By definition, if $y \in V(\mathcal{P}_j)$ and $x \preccurlyeq y$, then $L(y) \geqslant L(v)$. Conversely, if $y \in V(\mathcal{P}_j)$ and $L(y) = L(v) + h$ for $h \geqslant 0$, then $y = f^h(v)$; then $x \preccurlyeq v \preccurlyeq y$, and the result holds by transitivity. □

To determine the supremum function for all vertices, it will be helpful to be able to efficiently determine which path of $\mathcal{C}$ a given vertex belongs to and how far it is from the initial vertex for it's path. Algorithm 4 describes a simple procedure to do this, which also produces the successor function for the path cover $\mathcal{C}$. (In the case where $|I| = |O|$, every path of $\mathcal{C}$ has an initial point in I; we then take $K = I$ to be the index set of the paths of $\mathcal{C}$.)

3.2.3 Detecting vicious circuits with respect to $\mathcal{C}$

If the influencing digraph $\mathcal{I}_f$ contains non-trivial di-connected components, we know that there are closed influencing walks — i.e. vicious circuits — for $\mathcal{C}$ in (G, I, O). In that case, Theorem 8 together with Theorem 9 imply that (G, I, O) has no causal flow, in which case we may as well abort. Recall that `SimpleTC` keeps track of di-connected components by allocating vertices to a component C after the elements of C have been completely determined. However, the state of being allocated into a component can be replaced in this analysis by *any* status of the vertex which is changed after the descendants of a vertex have been determined; and this status may be used to determine if a vicious circuit has been found.

Algorithm 5 is a simple procedure to initialize an array `status` over $V(G)$. A status of `none` will indicate that no descendants of the vertex have been determined (except itself), `fixed` will indicate that all descendants of the vertex have been determined, and `pending` will indicate that the descendants are in the course of being determined. Because output vertices have only themselves for descendants, their status is initialized to `fixed`; all other vertices are initialized with $\texttt{status}(v) = \texttt{none}$. At the same time, Algorithm 5 initializes a supremum function which represents only the relationships of each vertex to the ones following it on the same path.

Algorithm 4: `GetChainDecomp`$(G, I, O, \mathcal{C})$ — obtain the successor function f of $\mathcal{C}$, and obtain functions describing the chain decomposition of the influencing digraph $\mathcal{I}_f$

Require: (G, I, O) be a geometry with $|I| = |O|$
Require: $\mathcal{C}$ a path cover of (G, I, O)

let $P : V(G) \longrightarrow I$ an array
let $L : V(G) \longrightarrow \mathbb{N}$ an array
let $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$ an array
for all $i \in I$ **do**
  let $v \leftarrow i\,,\ \ell \leftarrow 0$
  while $v \notin O$ **do**
    $f(v) \leftarrow$ `next`$(\mathcal{C}, v)$
    $P(v) \leftarrow i\,;\ L(v) \leftarrow \ell$
    $v \leftarrow f(v)$
    $\ell \leftarrow \ell + 1$
  end while
  $P(v) \leftarrow i\,;\ L(v) \leftarrow \ell$
end for
return (f, P, L)

Algorithm 5: `InitStatus`(G, I, O, P, L) — initialize the supremum function, and the status of each vertex

Require: (G, I, O) is a geometry
Require: $P : V(G) \longrightarrow I$ maps each $x \in V(G)$ to $i \in I$ such that x is in the orbit of i under f
Require: $L : V(G) \longrightarrow \mathbb{N}$ maps each $x \in V(G)$ to $h \in \mathbb{N}$ such that $x = f^h(P(x))$

let $\mathtt{sup} : I \times V(G) \longrightarrow \mathbb{N}$ an array
let $\mathtt{status} : V(G) \longrightarrow \{\mathtt{none}, \mathtt{pending}, \mathtt{fixed}\}$ an array
for all $v \in V(G)$ **do**
  for all $i \in I$ **do**
    if $i = P(v)$ **then** $\mathtt{sup}(i, v) \leftarrow L(v)$
    else $\mathtt{sup}(i, v) \leftarrow |V(G)|$
  end for
  if $v \in O$ **then** $\mathtt{status}(v) \leftarrow \mathtt{fixed}$
  else $\mathtt{status}(v) \leftarrow \mathtt{none}$
end for
return $(\mathtt{sup}, \mathtt{status})$

3.2.4 An efficient algorithm for computing the natural pre-order of f

Algorithms 6 and 7 below represent a modified version of Algorithm 3, specialized to the application of computing the natural pre-order for the successor function f of a path cover $\mathcal{C}$. Rather than explicitly constructing the influencing digraph $\mathcal{I}_f$ and traversing directed walks in $\mathcal{I}_f$ (as is done in Algorithm 3), we instead traverse influencing walks for $\mathcal{C}$ (characterized by its' successor function) in the graph G.

Theorem 24. *Let f be a successor function of a path cover $\mathcal{C}$ for a geometry (G, I, O). Let $P : V(G) \longrightarrow I$ map vertices v to the initial point of the path of $\mathcal{C}$ that covers v, and let $L : V(G) \longrightarrow \mathbb{N}$ map vertices v to the integer $h \in \mathbb{N}$ such that $v = f^h(P(v))$. Then* `ComputeSuprema` *halts on input (G, I, O, f, P, L). Furthermore, let $\sigma =$* `ComputeSuprema`*(G, I, O, f, P, L). If $\sigma =$* `fail`*, then (G, I, O) does not have a causal flow; otherwise, (G, I, O) does have a causal flow, and σ is a supremum function $\mathtt{sup} : I \times V(G) \longrightarrow \mathbb{N}$ satisfying*

$$x \preccurlyeq y \quad \Longleftrightarrow \quad \mathtt{sup}(P(y), x) \leqslant L(y) \tag{5}$$

for all $x, y \in V(G)$, where $\preccurlyeq$ is the natural pre-order for f.

Proof — We will reduce the correctness of Algorithms 6 and 7 to that of Algorithm 3, where $D = \mathcal{I}_f$ is the

Algorithm 6: `TraverseInflWalk`$(G, I, O, f, \texttt{sup}, \texttt{status}, v)$ — compute the suprema of v and all of its' descendants, by traversing influencing walks from v

Require: (G, I, O) is a geometry
Require: $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$ is a successor function for (G, I, O)
Require: $\texttt{sup} : I \times V(G) \longrightarrow \mathbb{N}$
Require: $\texttt{status} : V(G) \longrightarrow \{\texttt{none}, \texttt{pending}, \texttt{fixed}\}$
Require: $v \in O^{\mathsf{c}}$

1: $\texttt{status}(v) \leftarrow \texttt{pending}$
2: **for all** $w = f(v)$ **and for all** $w \sim f(v)$ **do**
3:   **if** $w \neq v$ **then**
4:     **if** $\texttt{status}(w) = \texttt{none}$ **then** $(\texttt{sup}, \texttt{status}) \leftarrow \texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, w)$
5:     **if** $\texttt{status}(w) = \texttt{pending}$ **then**
6:       **return** $(\texttt{sup}, \texttt{status})$
7:     **else**
8:       **for all** $i \in I$ **do**
9:         **if** $\texttt{sup}(i, v) > \texttt{sup}(i, w)$ **then** $\texttt{sup}(i, v) \leftarrow \texttt{sup}(i, w)$
10:       **end for**
11:     **end if**
12:   **end if**
13: **end for**
14: $\texttt{status}(v) \leftarrow \texttt{fixed}$
15: **return** $(\texttt{sup}, \texttt{status})$

Algorithm 7: `ComputeSuprema`(G, I, O, f, P, L) — obtain the successor function f of $\mathcal{C}$, and compute the natural pre-order of f in the form of a supremum function and functions characterizing $\mathcal{C}$

Require: (G, I, O) is a geometry with $|I| = |O|$ and successor function $f : O^{\mathsf{c}} \longrightarrow I^{\mathsf{c}}$
Require: $P : V(G) \longrightarrow I$ maps each $x \in V(G)$ to $i \in I$ such that x is in the orbit of i under f
Require: $L : V(G) \longrightarrow \mathbb{N}$ maps each $x \in V(G)$ to $h \in \mathbb{N}$ such that $x = f^h(P(x))$

let $(\texttt{sup}, \texttt{status}) \leftarrow \texttt{InitStatus}(G, I, O, P, L)$
for all $v \in O^{\mathsf{c}}$ **do**
  if $\texttt{status}(v) = \texttt{none}$ **then** $(\texttt{sup}, \texttt{status}) \leftarrow \texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, v)$
  if $\texttt{status}(v) = \texttt{pending}$ **then return** $\texttt{fail}$
end for
return $\texttt{sup}$

digraph provided as the the input of the main procedure. Throughout, $\preccurlyeq$ denotes the natural pre-order of f.

For distinct vertices $v, w \in V(G)$, because $(v \to w) \in A(\mathcal{I}_f)$ if and only if either $w = f(v)$ or $w \sim f(v)$, we may replace the iterator limits "w **such that** $(v \to w) \in A(D)$" of the **for** loop starting on line 6 of Algorithm 3 with a loop iterating over $w = f(v)$ and $w \sim f(v)$: this is what we have on line 2 of `TraverseInflWalk`.

At line 8 of Algorithm 3, if $\texttt{Comp}(w) = \texttt{nil}$, we infer that v and w are in a common di-connected component of the digraph $\mathcal{I}_f$: this implies that $v \preccurlyeq w$ and $w \preccurlyeq v$. If $v \neq w$, this implies that $\preccurlyeq$ is not antisymmetric, and thus not a partial order; by Theorem 8, $\mathcal{C}$ is then not a causal path cover. We proceed by cases:

- If $\preccurlyeq$ is antisymmetric, then the influencing digraph is acyclic, in which case $\mathcal{I}_f$ has only trivial di-connected components. In this case, the following changes preserve the functionality of Algorithm 3:
  - — In the case that $w = v$ in the **for** loop, all the operations performed are superfluous, in which case we may embed lines 7 through 9 in an **if** statement conditioned on $w \neq v$.
  - — Because each vertex is the only vertex in its' component when $\mathcal{I}_f$ is acyclic, we may replace lines 11 through 19 of `SimpleTC` with a line setting $\texttt{Comp}(v)$ to an arbitrary non-`nil` value, which in this case may be interpreted as allocating the vertex v to its' di-connected component (i.e. the singleton $\{v\}$).

Also, the condition of line 8 is never satisfied in a call to $\texttt{SimpleTC}(v)$. Then, we may replace the conditional code with an arbitrary statement, e.g. a command to abort the procedure.

— After the above replacement, the value of `stack` is not used within the procedure call $\texttt{SimpleTC}(v)$, and has the same value after the procedure call to $\texttt{SimpleTC}(v)$ as it does before the call. Then, `stack` is superfluous to the performance of the algorithm. Similarly, the value of $\texttt{Root}(v)$ is not affected except to initialize it. We may then eliminate all references to either one.

— The value of $\texttt{Comp}(w)$ is only tested to determine whether or not it is `nil`, so we may replace the array `Comp` with `status`, and its' possible states of being `nil` or non-`nil` with the states of being `pending` and non-`pending`. We define the two values `none` and `fixed` to represent being non-`pending` and also having not yet been visited, and being non-`pending` and having been visited, respectively.

— Using the array `sup` to implicitly represent the sets of descendants, we may replace the union performed on line 9 with code which sets $\texttt{sup}(i, v)$ to the minimum of $\texttt{sup}(i, v)$ and $\texttt{sup}(i, w)$ for each $i \in I$. Note also that, because x is a descendant of v in $\mathcal{I}_f$ iff $v = x$ or x is a descendant of w, we may remove the initialization of $\texttt{Desc}(v)$ on line 5 of Algorithm 3 if we initialize `sup` for each vertex so that it represents each vertex as a descendant of itself (for instance, in the main procedure, which is replaced by `ComputeSuprema`).

By performing the substitutions described above, we can easily see that `TraverseInflWalk` together with `ComputeSuprema` is equivalent to Algorithm 3 when $\preccurlyeq$ is anti-symmetric. Then, $\sigma \neq \texttt{fail}$ because line 4 of `ComputeSuprema` is never evaluated; we then have $\sigma = \texttt{sup}$ as in Equation 5, from the correctness of Algorithm 3.

- If $\preccurlyeq$ is not antisymmetric, then there are distinct vertices $x, y \in V(G)$ such that $x \preccurlyeq y \preccurlyeq x$, in which case x and y are in a non-trivial component in $\mathcal{I}_f$. Then, the **for** loop of `BuildCausalOrder` will eventually encounter a vertex v of which x and y are descendants.

  In the depth-first traversal of influencing walks performed in $\texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, v)$, eventually a directed cycle containing both x and y will be discovered. Without loss of generality, assume that the depth-first traversal starting from v visits x before y: then, the depth-first traversal will eventually uncover a walk of the form

$$v \;\to\; \cdots \;\to\; x \;\to\; \cdots \;\to\; y \;\to\; \cdots \;\to\; y' \;\to\; x \;\; .$$

  Then in the procedure call $\texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, y')$, line 5 will find $\texttt{status}(x) = \texttt{pending}$, as line 1 of the procedure call $\texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, x)$ has been executed while line 14 has not. Then, the procedure aborts by returning $\mathcal{R}$ without first changing the status of $\texttt{status}(y')$ from `pending`.

  It is clear that if w' depends on w, and if $\texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, w')$ aborts with $\texttt{status}(w') = \texttt{pending}$ during a procedure call $\texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, w)$, then the latter will also abort with $\texttt{status}(w) = \texttt{pending}$. By induction, we may then show that for $v \in V(G)$ for which x and y are descendants, $\texttt{TraverseInflWalk}(G, I, O, f, \texttt{sup}, \texttt{status}, v)$ will abort with $\texttt{status}(v) = \texttt{pending}$ in the **for** loop in `ComputeSuprema`.

  By the analysis of the case where $\preccurlyeq$ is antisymmetric, the status $\texttt{status}(v) = \texttt{pending}$ will only occur at line 4 of `ComputeSuprema` if $\preccurlyeq$ is not antisymmetric. If this occurs, $\sigma = \texttt{fail}$; as well, no causal path cover exists for (G, I, O) by Theorem 9, and thus no flow exists for (G, I, O) by Theorem 8.

Thus, $\sigma \neq \texttt{fail}$ iff $\preccurlyeq$ is a partial order; and when this occurs, by reduction to Algorithm 3, `sup` corresponds to the natural pre-order $\preccurlyeq$ in the sense of Equation 5. □

Run-time analysis. We may analyze the run-time of Algorithm 7 as follows. Let $n = |V(G)|$, $m = |E(G)|$, $k = |I| = |O|$, and d be the maximum degree of G. The time required to execute the **for all** loop starting on line 8 of `TraverseInflWalk` is $O(k)$; then, aside from the work done in recursive invocations to `TraverseInflWalk`, the time required to perform an invocation of `TraverseInflWalk` for a vertex v is $O(k \deg f(v))$. Because the

first invocation of `TraverseInflWalk` for a vertex v will change `status`(v) to something other than `none`, which prevents any further invocations for v, `TraverseInflWalk` will only be called once for any given vertex in the course of Algorithm 7. Then, summing over all vertices $v \in V(G)$, the amount of time required to perform the **for all** loop starting on line 2 of `ComputeSuprema` is $O(km)$. The time required by `InitStatus` to initialize `sup` and `status` is $O(kn)$; then, the overall running time of Algorithm 7 is $O(km)$.

3.2.5 A slightly more efficient algorithm for finding a causal order for f

If $\mathcal{C}$ is a causal path cover, it also is possible to find a causal order $\boxed{\preccurlyeq}$ compatible with f which differs from the natural pre-order for f, or determine that none exists, by recursively assigning integer "level" values to vertices rather than building the set of descendants. For example, one may construct a function $\lambda : V(G) \longrightarrow \mathbb{N}$ satisfying

$$\begin{aligned} \lambda(x) &= 0\,, && \text{if there are no influencing walks for } \mathcal{C} \text{ ending at } x; \\ \lambda(x) &= 1 + \max\left\{ \lambda(y) \;\middle|\; x = f(y) \;\text{ or }\; x \sim f(y) \right\}\,, && \text{otherwise.} \end{aligned} \tag{6}$$

Note that the set $S(x)$ of vertices y such that $x = f(y)$ or $x \sim f(y)$ are the initial points for any influencing walk for $\mathcal{C}$ with one segment which ends at x. By constructing the predecessor function $g = f^{-1}$ of $\mathcal{C}$ rather than the functions P and L in Algorithm 4, we can easily find all elements of $S(x)$ in G by visiting $g(z)$ for $z = x$ or $z \sim x$. Then, such a level function can be constructed by a Tarjan style algorithm similar to Algorithm 6, using the `status` array in the same way, but traversing the arcs of the influencing digraph $\mathcal{I}_f$ in the opposite direction as `TraverseInflWalk`. We may then define $x \boxed{\preccurlyeq} y \iff [x = y] \vee [\lambda(x) < \lambda(y)]$.

It is easy to see that the resulting partial order $\boxed{\preccurlyeq}$ resulting would have the same maximum-chain length as the natural pre-order $\preccurlyeq$: any maximal chain in $\preccurlyeq$ is a list of the end-points of consecutive segments in an influencing walk for $\mathcal{C}$, which will be a maximum chain in $\boxed{\preccurlyeq}$. However, $\boxed{\preccurlyeq}$ also contains relationships between vertices with no clear relation in the influencing digraph $\mathcal{I}_f$, because it suffices for two vertices to be on different "levels" for them to be comparable.

Such a causal order $\boxed{\preccurlyeq}$ can actually be constructed in $O(m)$ time, because the algorithm to construct it consists essentially of just a depth-first traversal with operations taking only constant time being done at each step. We have instead presented the above algorithm because the extra time required to obtain the coarsest compatible causal order for f will not affect the asymptotic run time of the complete algorithm for finding a flow, because of the immediate reduction to the well-studied problem of transitive closure, and in the interest of describing an algorithm to construct the natural pre-order for f (being the coarsest compatible causal order for f).

3.3 The complete algorithm

We now describe the complete algorithm to produce a flow for a geometry (G, I, O), using Algorithms 2 and 7.

Algorithm 8: `FindFlow`(G, I, O) — try to find a flow for (G, I, O)

Require: (G, I, O) is a geometry with $|I| = |O|$

let $\tau \leftarrow$ `BuildPathFamily`(G, I, O)
if $\tau =$ `fail` **then return** `fail`
let $(f, P, L) \leftarrow$ `GetChainDecomp`(G, I, O, τ)
let $\sigma \leftarrow$ `ComputeSuprema`(G, I, O, f, P, L)
if $\sigma =$ `fail` **then**
  return `fail`
else
  return (f, P, L, σ)
end if

Corollary 25. *Let (G,I,O) be a geometry with $|I| = |O|$. Then* `FindFlow` *halts on input (G,I,O). Furthermore, if* `FindFlow`$(G,I,O) =$ `fail`*, then (G,I,O) does not have a causal flow; otherwise,* `FindFlow`$(G,I,O) =$ $(f,P,L,\mathtt{sup})$*, and $(f,\preccurlyeq)$ is a causal flow, where $\preccurlyeq$ is characterized by*

$$x \preccurlyeq y \quad\Longleftrightarrow\quad \mathtt{sup}(P(y),x) \;\leqslant\; L(y)\,; \tag{7}$$

and is the natural pre-order for f.

Proof — By Corollary 19, a causal path cover exists for (G,I,O) only if `BuildPathFamily`(G,I,O) sets $\tau \neq$ `fail` on line 1; thus if $\tau =$ `fail`, (G,I,O) has no causal flow by Theorem 8. Otherwise, τ is a path cover. If `BuildCausalOrder` sets $\sigma =$ `fail` on line 4, (G,I,O) has no causal flow by Theorem 24. Otherwise, the relation $\preccurlyeq$ characterized by Equation 7 is the natural pre-order for f and a causal order, in which case $(f,\preccurlyeq)$ is a causal flow. □

Run-time analysis. Because $\tau \neq$ `fail` at line 1 implies that τ is a path cover, `GetChainDecomp` visits each vertex $v \in V(G)$ once to assign values for $P(v)$, $L(v)$, and possibly $f(v)$ in the case that $v \in O^{\mathsf{c}}$. Then, its' running time is $O(n)$. The running time of `FindFlow` is then dominated by `BuildPathCover` and `ComputeSuprema`, each of which take time $O(km)$.

4 Potential Improvements

This paper has described efficient algorithms for finding flows, with the aim of not requiring prior knowledge of graph-theoretic algorithms in the presentation. This constraint has led to choices in how to present the algorithms which may make them less efficient (in practical terms) than may be achievable by the state of the art; and no significant analysis of the graphs themselves have been performed. Here, I discuss issues which may allow an improvement on the analysis of this article.

4.1 Better algorithms for finding path covers

For network-flow problems (the usual tools used for solving questions of maximum-size collections of paths in graphs), there is a rich body of experimental results for efficient algorithms. However, there seems to be very little discussion in the literature of the special case where all edge capacities are equal to 1, which is relevant to the problem of finding maximum collections of vertex-disjoint $I - O$ paths. It is difficult to determine, in this case, whether there is a significant difference in the performance of various algorithms. Although it is less efficient than other algorithms for general network flow problems, the most obvious choice of network flow algorithm for finding a maximum family of vertex-disjoint $I - O$ paths is the Ford-Fulkerson algorithm, which has an asymptotic running time $O(km)$. This running time is identical to Algorithm 2: this should not be surprising, as Algorithm 1 essentially implements a depth-first variation of the Ford-Fulkerson algorithm for finding an augmenting flow.

A more thorough investigation of network flows may yield an improved algorithm for finding a path cover for (G,I,O), which (when coupled with the faster algorithm for finding a minimum-depth causal order) would yield a faster algorithm for finding causal flows.

4.2 Extremal results

Consider all the ways we can add edges between n vertices to get a geometry with k output vertices and a causal flow. Just to achieve a path cover, we require $n - k$ edges; this lower bound is tight, as graph consisting of just k vertex-disjoint paths on n vertices has this many edges, and the paths represent a causal path cover of that graph. The more interesting question is of how many edges are required to force a graph to not have any causal path covers.

Let $\tilde{n}$ be the residue of n modulo k. Consider a collection of k paths $\{P_j\}_{j \in [k]}$, given by $P_j = p_j^{(0)} \cdots p_j^{(\lceil n/k \rceil - 1)}$ for $j < \tilde{n}$, and $P_j = p_j^{(0)} \cdots p_j^{(\lfloor n/k \rfloor - 1)}$ for $j \geqslant \tilde{n}$. Then, let G be the graph defined by adding the edges $p_h^{(a)} p_j^{(a)}$ for all a and $h \neq j$ where these vertices are well-defined, and $p_j^{(a)} p_h^{(a+1)}$ for all a and $h < j$ where these vertices are well-defined. We may identify the initial point of the paths P_j as elements of I and end-points as elements of O: then, let $M(n,k)$ denote the geometry (G, I, O) constructed in this way.

The geometry $M(n,k)$ has the obvious successor function given by $f(p_j^{(a)}) = p_j^{(a+1)}$ for all j and a where both vertices are defined. Then, consider the natural pre-order for f:

- (i). we obviously have $p_j^{(a)} \preccurlyeq p_j^{(b)}$ for $a \leqslant b$, for every $j \in [k]$;
- (ii). from the edges $p_h^{(a)} p_j^{(a)}$, we obtain $p_h^{(a-1)} \preccurlyeq p_j^{(a)}$ for all $h, j \in [k]$ and all $a > 0$; and
- (iii). from the edges $p_h^{(b+1)} p_j^{(b)}$ for $h < j$, we obtain $p_h^{(b)} \preccurlyeq p_j^{(b)}$ and $p_j^{(b-1)} \preccurlyeq p_h^{(b+1)}$. (Note that the second of these two constraints is redundant, as $p_j^{(b-1)} \preccurlyeq p_h^{(b)} \preccurlyeq p_h^{(b+1)}$ is implied by the above two cases.)

Then, the natural pre-order $\preccurlyeq$ on $M(n,k)$ is closely related to the lexicographical order on ordered pairs: $p_h^{(a)}$ and $p_j^{(b)}$ are incomparable if they are both endpoints of their respective paths P_h and P_j, and otherwise $p_h^{(a)} \preccurlyeq p_j^{(b)}$ if and only if either $a < b$, or $a = b$ and $h \leqslant j$. This is clearly a partial order, so $M(n,k)$ has a causal flow: and it has $kn - \binom{k+1}{2}$ edges in total.

I conjecture that this is the maximum number of edges that a geometry on n vertices with k output vertices can have. If this can be proven, we can determine that certain geometries have no flows just by counting their edges; the upper bounds of this paper can then be improved to $O(k^2 n)$.

5 Open Problems

To conclude, I re-iterate the open problems presented in [7].

1. **The general case.** When $|I| > |O|$, it is easy to see that a causal flow cannot exist, because no successor function f may be defined. This leaves the case where $|I| < |O|$. If $\delta = |O| - |I|$, we may test sets $\partial I \subseteq I^{\mathsf{c}}$ with $|\partial I| = \delta$ to see if the geometry $(G, I \cup \partial I, O)$ has a causal flow: doing this yields an $O(kmn^\delta)$ algorithm for finding a causal flow for (G, I, O). Is there an algorithm for finding causal flows in an arbitrary geometry with $|I| \leqslant |O|$, whose run-time is also polynomial in $\delta = |O| - |I|$?

2. **Graphs without designated inputs/outputs.** Quantum computations in the one-way model may be performed by composing three patterns: one pattern to prepare an appropriate quantum state, a pattern to apply a unitary that state (in the vein that we have been considering in this article), and a final pattern which measures the resulting state in an appropriate basis. The composite pattern has no input or output qubits, and so has only the measurement signals as an output. The result of the computation would then be determined from the parity of a subset of the measurement signals.

   Given a graph without any designated input or output vertices, what constraints are necessary to allow a structure similar to a causal flow to be found, which would guarantee that deterministic n qubit operations in the sense of [5] can be performed in the one-way measurement model with the entanglement graph G?

3. **Ruling out the presence of causal flows with only partial information about G.** Are there graphs G where it is possible to rule out the presence of a flow for (G, I, O) from a proper sub-graph of G, or given $n = |V(G)|$, $m = |E(G)|$, and $k = |I| = |O|$? (This question obviously includes the extremal problem asked earlier.)

4. **Relaxing the causal flow conditions for Pauli measurements.** Suppose that, in addition to I and O, we know which qubits are to be measured in the X axis and which are to be measured in the Y axis (corresponding to measurement angles 0 and $\pi/2$ respectively). These qubits can always be measured first in a pattern, by absorbing byproduct operations on those qubits and performing signal shifting. However, the analysis of patterns in terms of causal flows does not take this into account, as it is independent of

measurement angles. Is it possible to develop a natural analogue for causal flows which represents these qubits as minimal in the corresponding causal order, which may be efficiently found for geometries with $|I| = |O|$ or $|I| \leqslant |O|$ generally?

The results of this article were inspired by the similarity between of the characterization in terms of causal flows, with aspects of graph theory related to Menger's Theorem in general, and the relationship between influencing walks and alternating walks in particular. Investigation into open questions involving efficient construction of causal flows or relaxations of them may benefit from additional investigation of this link.

Acknowledgements. I would like to thank Elham Kashefi, who interested me in the problem of efficiently finding causal flows for geometries and for helpful discussions; Donny Cheung and Anne Broadbent, who provided useful insights; and Rob Raussendorf, for his feedback on the presentation of this paper.

6 Acknowledgements

I would like to thank Elham Kashefi, who interested me in the problem of efficiently finding causal flows for geometries and for helpful discussions; Donny Cheung, who provided helpful remarks on sufficient conditions for a causal flow to exist; and Anne Broadbent and Rob Raussendorf, for their contributions to my understanding of the problem.

References

[1] R. Raussendorf, H.-J. Briegel. *A one-way quantum computer.* Physical Review Letters vol. 86 (2001), pp. 5188–5191.

[2] R. Raussendorf, H.-J. Briegel. *Computational model underlying the one-way quantum computer.* Quantum Information and Computation vol. 2 #6 (2002), pp. 443–486.

[3] R. Raussendorf, D. E. Browne, H.-J. Briegel. *Measurement-based quantum computation on cluster states.* Physical Review A vol. 68, article 022312 (2003).

[4] N. de Beaudrap. *Characterizing & constructing flows in the one-way measurement model in terms of disjoint I–O paths.* arXiv:quant-ph/0603072v2 (2006).

[5] V. Danos, E. Kashefi. *Determinism in the one-way model.* arXiv:quant-ph/0506062 (2005).

[6] N. de Beaudrap, V. Danos, E. Kashefi. *Phase map decompositions for unitaries.* arXiv:quant-ph/0603266 (2006).

[7] N. de Beaudrap. *Finding flows in the one-way measurement model.* arXiv:quant-ph/0611284 (2006).

[8] V. Danos, E. Kashefi, P. Panangaden. *The Measurement Calculus.* arXiv:quant-ph/0412135 (2004).

[9] R. Diestel. *Graph Theory.* 3rd ed. Springer-Verlag Heidelberg, New York (2005).

[10] J. Clark, D. A. Holton. *A first look at graph theory.* World Scientific Publishing, Singapore (1991).

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms,* 2nd ed. MIT Press and McGraw-Hill (2001). Section 26.2, pp. 651–664.

[12] W. J. Cook, W. H. Cunningham, W. R. Pulleybank, A. Schrijver. *Combinatorial Optimization.* Wiley-Interscience New York (1998). Section 3.2, pp. 38–45.

[13] E. Nuutila. *Efficient Transitive Closure Computation in Large Digraphs.* Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series #74, Helsinki (1995). See also `http://www.cs.hut.fi/~enu/thesis.html`.